Configuration

Papers from the Configuration Workshop at IJCAI'05

Dietmar Jannach
Alexander Felfernig
Chairs

July 30, 2005

19th International Joint Conference on Artificial Intelligence

University of Edinburgh
Edinburgh – Scotland

Organizing Committee

Claire Bagley. Oracle Corporation, USA.
Timo Soininen. Helsinki University of Technology, Finland.
Markus Stumptner, University of South Australia, Australia.

Program Committee

Michel Aldanondo, Centre de Genie Industriel, Ecole des Mines d'Albi, France.
Claire Bagley, Oracle Corporation, USA.
Boi Faltings, Swiss Federal Institute of Technology, Switzerland.
Gerhard Friedrich, University Klagenfurt, Austria.
Esther Gelle, ABB Switzerland, Corporate Research, Switzerland.
Albert Haag, SAP, Germany.
Ulrich Junker, ILOG S.A., France.
Diego Magro, Universita di Torino, Italy.
Michael Koch, Technical University Munich, Germany.
Daniel Mailharro, ILOG S.A., France.
Barry O'Sullivan, University College Cork, Ireland.
Klas Orsvarn, Tacton System AB, Sweden.
Frank Piller, TUM Research Center Mass Customization & Customer Integration, Germany.
Marty Plotkin, Oracle Corporation, USA.
Timo Soininen, Helsinki University of Technology, Finland.
Pietro Torasso, Universita di Torino, Italy.
Markus Stumptner, Advanced Computing Research Center, Australia.
Markus Zanker, University Klagenfurt, Austria.

Contents

1. **A Decision Tree Learning and Constraint Satisfaction Hybrid for Interactive Problem Solving**
   Barry O'Sullivan and Alex Ferguson and Eugene C. Freuder..... p. 1
2. **Linear Functions for Interactive Configuration Using Join Matching and CSP Tree Decomposition**
   Sathiamoorthy Subbarayan, Henrik Reif Andersen..... p. 7
3. **Debugging User Interface Descriptions of Knowledge-based Recommenders**
   Alexander Felfernig, Shchekotykhin Kostyantyn..... p. 13
4. **A Conceptual Model for Configurable Services**
   Mikko Heiskala, Juha Tiihonen, and Timo Soininen..... p. 19
5. **Modeling Multiple System Families**
   Esther Gelle..... p. 25
6. **Design Space Exploration Using Constraint Satisfaction**
   Noel Titus, Karthik Ramani..... p. 31
7. **Configuring Loopholes: Interactive Consistency Maintenance of Business Rules**
   Markus Stumptner, Michael Schrefl..... p. 37
8. **Interactive Configuration and Evaluation of a Heat Treatment Operation**
   M. Aldanondo, E. Vareilles, K. Hadj-Hamou and Paul Gaborit..... p. 40
9. **STAR-IT: a Tool to Build STAR Applications**
   Diego Magro..... p. 46
10. **Kumbang Configurator–A Configuration Tool for Software Product Families**
    V. Myllärniemi and T. Asikainen and T. Männistö and T. Soininen..... p. 51
11. **PLM-integrated Configurators for Machine and Plant Construction**
    Philipp Ackermann..... p. 57
12. **Co-Configuration of Products and On-Line Service Manuals**
    Carsten Sinz and Wolfgang Küchlin..... p. 60
13. **Reconfiguration – A Problem in Search of Solutions**
    Peter Manhart..... p. 64
14. **"Dealing" with Configurable Products in the SAP Business Suite**
    Albert Haag..... p. 68
15. **Configurators in innovative or standardized business processes**
    Gerhard Fleischanderl..... p. 72
16. **MCml2**
    Jiri Vyskocil, Martin Trcka, Libor Denner, Petr Svab..... p. 73
17. **Tacton Configurator – Research directions**
    Klas Orsvarn..... p. 75

Preface

Configuration problems have always been subject of interest for the application and the development of advanced AI-based techniques. In particular, the great variety and the complexity of configurable product models both require powerful knowledge-representation formalisms while on the other hand efficient reasoning methods are needed to provide intelligent interactive behavior including solution search, satisfaction of user preferences, personalization and optimization.

Driven by academic and industrial interest, the IJCAI'05 Configuration Workshop now continues a series of successful workshops that started at the AAAI'96 Fall Symposium and continued on IJCAI, AAAI, and ECAI since 1999. The selection of papers of this year's workshop demonstrates the wide range of applicable AI techniques in the domain including contributions both on algorithms, search performance and user interaction, as well as on novel approaches related to knowledge-representation and product modeling.

Beside the research aspect, this year's workshop also has a strong emphasis on real-world configuration problems, tools, and applications. The working notes therefore also include a selected set of short papers and position statements from industry describing experience reports, problem statements, and advanced configuration tools. In addition, the workshop also features a dedicated "industry session" and a discussion panel with participants from major configurator vendors as well as representatives from companies with long-term experiences in deploying configuration applications in different domains. As such, this year's event aims at providing a stimulating environment for knowledge-exchange among academia and industry and thus building a solid basis for further developments in the field.

Dietmar Jannach
Alexander Felfernig

University Klagenfurt, Austria

A Decision Tree Learning and Constraint Satisfaction Hybrid for Interactive Problem Solving*

Barry O'Sullivan and **Alex Ferguson** and **Eugene C. Freuder**
Cork Constraint Computation Centre
Department of Computer Science, University College Cork, Ireland
{b.osullivan|a.ferguson|e.freuder}@4c.ucc.ie

Abstract

In this paper we present an approach to using decision tree learning as a basis for improving search performance for real-world constraint satisfaction problems. The problem used to evaluate the approach is a large real-world configuration benchmark from the automotive industry.

1 Introduction

The traditional focus of the constraint satisfaction community is solving a CSP instance once, and once only. For example, finding a solution to a set of layout constraints, a planning problem, a scheduling problem or timetabling problem. However, in many real-world domains CSPs are solved many times over. For example, a CSP may represent a configuration problem. Since different users will seek different configurations, the same instance of the configuration CSP is solved repeatedly, but the users' choices change each time. In such a scenario users make decisions by interactively assigning values to some subset of the variables of the underlying CSP, representing a set of choices or desires. The task of a system reasoning about such a problem, or *configurator*, is to present a solution that satisfies the set of choices the user has made, or to report that none exists.

In real-world application contexts, such as the one outlined above, the underlying CSP often exhibits a structured solution space. For example, there may be many solutions to the problem that are different only in terms of a small number of assignments. Specifically, many real-world problems exhibit some form of interchangeability amongst values [Freuder, 1991]; i.e. we may be free to change one assignment in a solution, holding the others fixed, and still have a solution. In such cases machine learning techniques can be employed to take advantage of this structure in order to enhance the performance of the constraint solver used to solve the CSP. It is this observation that underpins the work we present here.

In many domains, such as product configuration, users are better able to specify their requirements by critiquing a solution. For example, a user might specify a car with a sunroof and air-conditioning, but having been presented with a car that met these requirements but having only two seats, realize that it was important, perhaps more important, to have five seats. Note that the user model used in this work in one in which the system takes the initiative in deciding which variables the user should make decisions about, but if this is regarded as a limitation it should only be seen as a limitation in finding the *first* solution. The user will be free to critique this solution on the basis of his/her preferences in a subsequent solution critiquing phase [Pu and Faltings, 2004]. Our objective is to help a user find an *initial solution* that satisfies some of his/her desires as quickly as possible so that the critiquing phase can begin.

The approach we present combines decision tree learning [Quinlan, 1986] with constraint satisfaction techniques. We induce a decision tree from a set of positive examples (solutions to the CSP) and negative examples (non-solutions to the CSP). Positive examples can be found either by search or, as is more likely in many real-world contexts such as configuration, from a catalogue or brochure containing examples of the solutions that are available. It is trivial to randomly generate a set of negative examples. The decision tree acts as a first step in a two step solving process, specifying subproblems of the CSP to be searched using standard techniques. The improvements that can be obtained are often considerable.

We propose a number of different ways to post-process the decision tree that have the potential to dramatically reduce the search effort required to find a solution. Often search is unnecessary, since it can be proven by the decision tree alone that a solution exists or does not exist. In terms of the overall result, we demonstrate that the approach yields significant reductions in search effort on a large real-world configuration benchmark when compared with a good constraint solver. Furthermore, we show that the reduction in search cost comes at negligible space cost.

We also present a surprising result: using decision trees induced from training sets of increasing size, the accuracy associated with classifying unseen solutions and non-solutions quickly becomes close to perfect. This is unexpectedly favourable, given that the decision trees we induce remain very small.

This paper builds on earlier work on this topic [O'Sullivan *et al.*, 2004]. This earlier work proposed using decision trees for improving upon the effort required to find a solution to

* This work has received support from Enterprise Ireland (Grant SC/2002/289) and Science Foundation Ireland under Grant No. 03/CE3/I405, as part of the Centre for Telecommunications Value-Chain-Driven Research (CTVR), and Grant 00/PI.1/C075.

a CSP. In particular, the earlier work demonstrated how problems with high levels of interchangeability could benefit from such an approach. In this paper a number of new results are presented. Firstly, we demonstrate the technique on a large real-world configuration problem. Secondly, we propose the notion of a lax query. Thirdly, we demonstrate that the classification accuracy of decision trees built for the real-world problem studied here is extremely high while the size of such decision trees is very small.

The remainder of this paper is organized as follows. Section 2 presents the details of our approach. In Section 3 we present results from an empirical evaluation carried out on a large real-world configuration problem. In Section 4 we briefly review the relevant related literature, demonstrating the novelty of the approach proposed here. In Section 5 a number of directions for future work are discussed and some concluding remarks are made.

2 A Hybrid Approach to Problem Solving

Decision tree learning is a standard machine learning technique for approximating discrete-valued functions [Quinlan, 1986]. Decision trees make classifications by sorting the features of an instance through the tree from the root to some leaf node. Consider the decision tree presented in Figure 1. At each internal node a test on some variable is performed, and each subtree corresponds to a possible outcome for that test. Upon reaching a leaf node, a classification is made. In Figure 1 there are three possible classifications: '+' indicating the classification 'solution', '-' indicating the classification 'non-solution', and '!' indicating the classification 'definite non-solution'. The distinction between '-' and '!' will be discussed in Section 2.2.

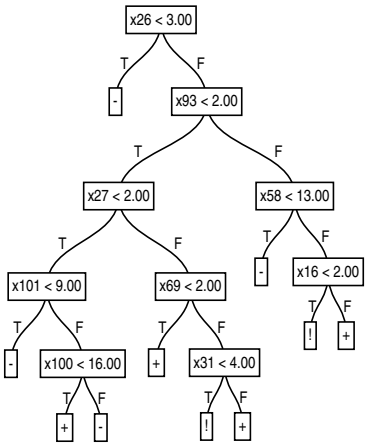


Fig. 1: Example decision tree.

2.1 The Core Idea

While decision trees are typically used for classification tasks, we propose that they can also be used in problem solving. In interactive problem solving, such as product configuration, the key idea is to be able to either find a solution to a set of user requirements, or prove quickly that none exists.

Our objective here is to use a hybrid approach that combines decision tree learning with standard constraint satisfaction techniques to improve the performance of an interactive constraint-based application. Based on a training set of examples drawn from a subset of the solutions and non-solutions to the CSP, a decision tree can be induced, which can then be used to classify future (test) instances, predicting whether they are or are not solutions on the basis of the prior instances.

We assume that the problem we wish to solve has a very large number of solutions, typically the case in configuration. For such problems, assuming the availability of a training set is not an issue, and can be found offline.

In addition, during an interactive search session, the decision tree can be used to select the next variable about which the user should be asked for a decision. Presented with a variable, v, a user can provide two types of response. Firstly, the user can specify a *tight* unary equality constraint assigning v to some value from its domains. Alternatively, the user can let the decision tree pose a question about the value of a variable by presenting the user with a choice between the single unary constraint on that node, and its negation. We refer to the set of user choices as a query. For example, in terms of Figure 1, we might construct a tight query, Q_t, such as:

$$\{(x_{26}, \{3\}), (x_{93}, \{1\}), (x_{27}, \{1\}), (x_{101}, \{9\})\}$$

or a lax query, Q_l such as:

$$\{(x_{26}, \{3\}), (x_{93}, \{1\}), (x_{27}, \{1\}), (x_{101}, \{9 \ldots 12\})\}$$

where the maximum value in the domain of x_{101} is 12. Note that both of these queries correspond to the same path through the tree.

While inviting the user to specify a lax unary constraint on a variable might not seem to be very useful, in many interactive applications users are not willing to provide precise assignments, but prefer to give some less specific requirements, having the system present solutions to them. On the basis of such solutions the user could begin to understand the nature of the solutions that are available, helping the user to refine his set of requirements.

In addition, while the query building process we have described is completely system-driven, the objective is to find a solution to the problem quickly. Once this solution is available, the user can use it as the basis for a critiquing phase in order to find the desired solution that satisfies the users preferences [Pu and Faltings, 2004]. As we will see from the evaluation, orders of magnitude speed-up in finding a solution can be achieved.

A query corresponds to a path through the decision tree starting from the root. There are two special cases to consider. Firstly, the user may not wish to answer every question generated by the decision tree. For example, both queries presented above place us at the node testing variable x_{100}. There are two possible paths remaining to classification nodes, providing us with alternative additional unary constraints that can be used to extend the query – in this case depending on whether we wish to consider the situation where x_{100} is less than 16, or greater than or equal to 16. Secondly, he may wish to provide additional unary constraints once there are no longer any decision tree nodes to use as the basis for prompting him. In

this case, the user's query will have reached a leaf node and additional unary constraints are presented. For example, the query $\{(x_{26}, \{1\}), (x_{40}, \{1\})\}$ is classified as negative by the tree on the basis of the unary constraint on variable x_{26}, however we also have an additional constraint on variable x_{40}.

Thus, in general, having followed a path through the decision tree, we obtain a set of leaf nodes, each associated with a number of constraints, and a positive/negative classification. At this point we can rely on constraint satisfaction to either find a solution satisfying the user's query constraints, or verify that none exists. This is done by solving, in turn, the subproblem associated with each *positive* classification, stopping as soon as we find one that is soluble; if we find none, then we repeat the process on the sub-problems associated with the *negative* classifications. This last step is necessary to achieve accurate results, as a presumptively negative region of the solution space may not in fact be solution-free, but simply lack a solution in the training set. Thus, in the case of the longer query, Q_t, presented earlier, we will first solve the single subproblem corresponding to the extension of the query: $Q_t \cup \{x_{100} < 16\}$ (classified as positive) and subsequently, if necessary, that corresponding to: $Q_t \cup \{x_{100} \geq 16\}$.

Therefore, having processed a query, we get advice from the decision tree that indicates whether or not a solution exists and a number of sub-problems that can be searched to guarantee completeness of the approach.

2.2 Enhancements

We have found that using a decision tree that has been induced directly from the training set does not improve over immediate search on the user's query. Generally we simply end up solving essentially the same problem, or several subproblems corresponding to a partition thereof. However, we can identify two distinct ways of enhancing the potential performance improvement possible in decision tree lookup.

Firstly, once the decision tree has been induced, we can analyze each negative node to determine if there is *any* extension of that node's path that leads to a solution in the problem at large. If there is not, we can label this node as a *definite negative* classification. For example, in Figure 1, the path corresponding to the query $\{(x_{26}, \{3\}), (x_{93}, \{1\}), (x_{27}, \{2\}), (x_{69}, \{2\}), (x_{31}, \{1\})\}$, has no solutions, and thus we can annotate the corresponding leaf node with a '!' rather than a '-'. The consequences of this are that we can entirely eliminate such nodes from consideration when we encounter them while processing a query, and that if only one such node is found, we may immediately terminate. If the sub-problem is in fact soluble, we leave the node's annotation unchanged, and treat it exactly as we previously treated all negatively classified nodes, i.e. this node will still have to be searched if it is encountered. We will refer to trees that have be processed in this way as being *purified* to reflect the fact that they identify all definite negative classifications, i.e. those that are pure with respect to the *whole* solution space, rather than simply a given training set.

Secondly, associated with each positive node, we can exemplify the corresponding partial solution by storing one complete solution which has the path to that node as a prefix. The consequence of this is that if a user's query does not extend past that classification node we can immediately stop at that point, saving us from performing any search. On the other hand, if the user's query does extend past a particular classification node, but the user's remaining constraints are consistent with the solution associated with that leaf, we can stop, again saving us from performing any search. We will refer to trees that have be processed in this way as being *exemplified*. We can combine both enhancements to form a *purified exemplified* decision tree.

3 Experiments

We ran experiments using the Renault Megane Configuration benchmark [Amilhastre *et al.*, 2002], a large real-world configuration problem. The variables represent various options such as engine type, interior options, etc. The problem consists of 101 variables, domain size varies from 1 to 43, and there are 113 constraints, many of them non-binary. The number of solutions to the problem is over 1.4×10^{12}.

The constraint solver we used in our experiments is a generalized forward checker, that propagates k-ary constraints when $k-1$ variables are instantiated. We used minimum ratio of domain over forward degree as a variable ordering heuristic, and lexicographically chose values from within each domain. In order to improve solution time, we added the binary projections of all non-binary constraints in the problem, giving us 422 constraints in total. Adding these binary projections improves the performance of generalized forward checking by several orders of magnitude in constraint checks.

This solver provided us with a baseline for comparing the performance of the decision tree-based hybrid against traditional direct search; and furthermore, having identified subproblems from the decision tree, gives a facility to solve those. Therefore to some extent, we can remove some possible bias from any comparison by using the same solver in each case, thus getting a relatively 'like with like' comparison.

Two different experiments were conducted. Firstly, we studied the classification accuracy of the decision trees we induced directly from the training set (Section 3.1). Secondly, we compared the performance of the decision tree-based hybrid against the base-line constraint solver (Section 3.2).

We generated decision trees based on different sized training sets comprising a number of exemplars of both 'classes' to be discriminated: solutions to the CSP (positive examples), and non-solutions — that is, fully instantiated tuples that are not solutions to the CSP. We have chosen, somewhat arbitrarily, to use throughout three times as many negative examples as positive ones. In partial justification we note that solutions are both less heavily represented in the space, and are more costly to obtain.

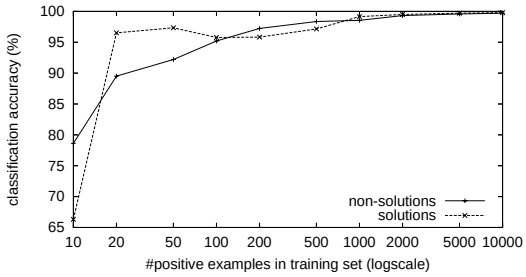
Solutions were generated by fixing a variable order, using which a set of feasible partial solutions (prefixes) of increasing length were calculated, until the cardinality of this prefix set exceeded the size of the desired solutions set. The desired number of prefixes were then selected and extended randomly to solutions.

Obtaining the negative examples is straightforward, as they represent the considerable majority of the solution space, and can, thus, easily be generated equiprobably at random. We constructed decision trees using ITI [Utgoff *et al.*, 1997]. Trees were built using the 'quick build' option, i.e. using the `-f` flag of the ITI program.

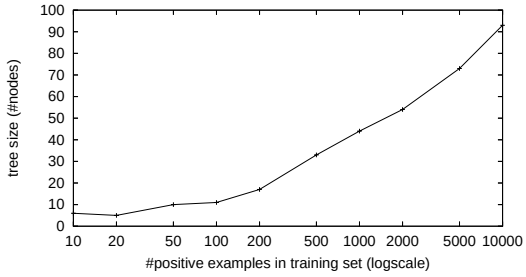
We conducted each experiment with decision trees induced from training sets containing 10, 20, 50, 100, 200, 500, 1000, 2000, 5000 and 10000 positive examples, in turn.

3.1 Experiment #1: Classification Accuracy

The objective of this experiment was to determine the accuracy with which a decision tree can classify unseen complete instantiations of all the variables in the problem. Two test-sets were created, one containing solutions only, the other containing non-solutions only, each built similarly to the training sets described earlier. However, the test-sets were significantly larger than the training sets, and were filtered so as to be strictly disjoint with each, in order to avoid any possible testing bias. Test-sets with approximately 28,000 positive and 90,000 negative instances were generated. Each instantiation in our test-sets was classified by each of the trees, and the results plotted separately based on the type of test-set in Figure 2(a). The size of each tree was measured in terms of the number of leaf nodes and plotted in Figure 2(b).



(a) Classification accuracy.



(b) Size of the decision tree.

Fig. 2: Classification accuracy results.

From Figure 2 it is clear that the classification accuracy of the tree rapidly becomes exceptionally high for both test sets. Based on decision trees induced from a training-set of 100 solutions we are able to achieve classifications accuracies in excess of 95%. Based on tree induced from 1,000 solutions we achieve an accuracy of over 99% and for 10,000 solutions we achieve over 99.9%. Given the size of the problem, the number of solutions and the size of the decision trees that have been induced (Figure 2(b)), this is an extremely encouraging result.

Given that the decision tree is relatively small, involving only tens of nodes, the decision tree represents a much smaller representation than the underlying CSP by a significant factor. The original CSP, represented using 113 extensional constraints, requires a file-size of 6.6Mb. As an automaton, the problem can be represented using 236,160 states and 306,809 transitions, which can be described in a file of 3.4Mb [Amilhastre *et al.*, 2002]. The small (maximum file size 23kb), but accurate, representation of the state space of the underlying problem that can be achieved using a decision tree is useful in many application domains, such as embedded systems, where an accurate, but compact, representation, of a potentially large problem is required.

3.2 Experiment #2: Improving Search

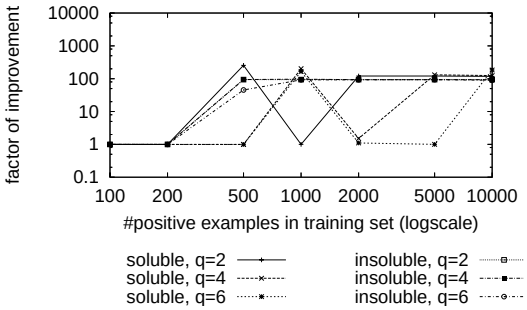
The objective of this experiment was to determine the utility of the decision tree hybrid as a basis for supporting interactive search based on queries presented by a simulated user.

The tree was used to build queries of various lengths. Queries are generated by following a path through the tree, randomly selecting a unary constraint at each test node up to the desired length. In the case of a tight query, each unary constraint is an assignment. In the case of a lax query, each unary constraint is an inequality. If the path is exhausted, random unary constraints are selected; for lax queries a non-singleton domain was chosen with a random 'split point'. We measure query length according to the number of distinct such constraint terms. In the tight case, this necessarily corresponds to a the same number of distinct variables; for lax queries, the same variable may recur in successively tighter constraints.

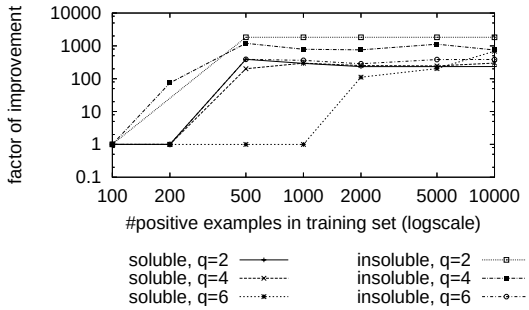
Using the hybrid approach, we look up the decision tree following the constraints in the query, then considering all sub-problems associated with the remaining reachable leaf nodes. Recall, that given a set of sub-problems to be considered, we first focus on solving those associated with positively classified leaves, then (indefinite) negatively classified ones.

In this experiment we studied four different decision tree and lookup configurations. Firstly, we considered the decision tree induced from each training-set without any of the enhancements discussed earlier; we will refer to this as the *impure tree*. Secondly, we considered the purified decision tree, and thirdly, the purified exemplified tree. In each of the above cases we considered their performance when we use tight queries. Lastly, we considered the performance of the purified exemplified trees when using lax queries. We only focus on the results obtained from the purified exemplified tree using both tight and lax queries, since no benefit was found using the impure tree and only improvements on processing insoluble queries are observed when definite negative classifications have been identified.

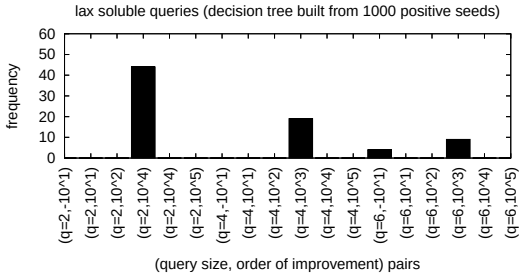
Queries of lengths 2, 4, and 6 were used, and for each of these 100 such queries were generated (very long queries quickly make the resultant sub-problems overwhelmingly insoluble, due to the high discrimination of the decision tree tests). For each query we computed the search effort re-



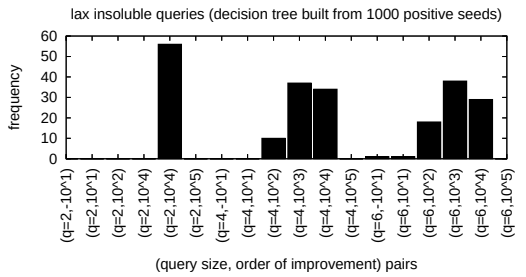
(a) Purified exemplified tree (Tight queries).



(b) Purified exemplified tree (Lax queries).



(c) Detailed improvement results for soluble queries.



(d) Detailed improvement results for insoluble queries.

Fig. 3: Results for constraint satisfaction.

quired by the reference solver to decide the user's query. This was compared with the search effort required by the hybrid decision-tree-based approach. Search effort is measured in terms of constraint checks, plus decision tree checks for the hybrid approach. We compute the improvement factor of the hybrid approach over the reference solver using cc_s/cc_h, where cc_s is the number of constraint checks required by the reference solver, and cc_h is the sum of the number of constraint checks and decision tree checks required by the hybrid approach.

In Figure 3, we show the median improvements associated with processing both soluble and insoluble queries separately using tight queries (Figure 3(a)) and lax queries (Figure 3(b)) on the pure exemplified decision tree.

It is clear that we obtain improvements for both soluble and insoluble queries. Once all definite negative classifications have been identified, we obtain at least two orders of magnitude for the larger trees when processing insoluble queries (Figure 3(a) and 3(b)). This improvement is essentially due to our not having to resort to any search at all on many insoluble queries, i.e. the decision tree is sufficient to determine insolubility, while the reference solver must perform some work to reach the same conclusion.

The most dramatic improvement can be observed with lax queries. In the case of soluble lax queries we can find that improvements of a factor of 300 are possible, while performance for insoluble queries is further improved by a marked amount. This is to be expected, as such queries constrain the original problem much less, and non-singleton unary constraints are not propagated as much as the assignments of tight queries by the reference solver. Therefore search is more expensive in such cases, and thus there is more room for improvement.

In Figures 3(c) and 3(d) we present, in more detail, the improvements achieved when processing lax queries on the pure exemplified tree, built from a training-set containing 1000 solutions, when processing both soluble and insoluble queries, respectively. Each graph shows the number of times we achieve an improvement of a given magnitude for different query sizes. Note that we rarely observe disimprovements of an order of magnitude (-10), but often observe improvements up to 10^4. These plots show the results of processing 100 queries of each size, therefore the frequencies for a given query size in both graphs will sum to 100. Taking a look at the level of individual queries (Figure 3(c) and 3(d)) we can see that there is a much more systematic improvement associated with processing insoluble queries.

We can see that using our hybrid approach never has a large negative impact on performance; Figures 3(c) and 3(d) show only a tiny negative impact for query sizes of length 6 in each case. This impact is not sufficient to affect the median improvement factor results presented in Figures 3(a) and 3(b). However, the hybrid approach can dramatically improve performance by orders of magnitude, particularly if users build lax queries. Such queries may be more likely to be used by

users who are not very familiar with the solution space, and who may be more confident giving less committal responses to the configurator. In such cases, the decision tree method excels.

4 Related Work

One of the significant challenges of constraint satisfaction techniques is the amount of search effort that may be required to solve a problem. In the general case, finding a solution to a CSP is NP-complete. To address this, many techniques have been developed that try to minimize or avoid costly backtracking during search.

It has long been known that tree-structured constraint problems can be solved without backtracking [Freuder, 1982]. This result has been extended to more general constraint networks also [Freuder, 1990; Dechter and Pearl, 1987]. However, these latter approaches may be impractical due to their exponential worst-case space complexity.

Recent work has combined automata theory and constraint satisfaction as a basis for supporting interactive configuration [Amilhastre *et al.*, 2002]. That approach essentially compiles a CSP into a form that can be used to answer particular questions without backtracking. Again, the disadvantage here is that the size of the automaton can grow exponentially, depending on the properties of the constraint problem being considered.

The notion of a general purpose backtrack-free representation for a CSP has been studied in the context of configuration [Beck *et al.*, 2004]. This approach uses a fixed variable ordering to preprocess a CSP. By working in reverse through the variable ordering, values that could cause a backtrack are removed, so that backtracking never occurs when the CSP is being solved 'online'. The transformed CSP is called a backtrack-free representation (BFR). Of course, when values are removed, so may be solutions. So in this approach the improvements in search efficiency come at the expense of lost solutions. While this is not acceptable in many domains, it is reasonable in some others where a solution needs to be found very quickly.

An alternative approach to improving the performance of search is the notion of nogood recording [Schiex and Verfaillie, 1994]. Nogoods are justifications of why particular instantiations of variables are inconsistent. By recording nogoods during search, backtracking can be minimized by exploiting this knowledge as search continues. However, in its general form this approach also requires exponential space, in the worst-case.

In this paper an approach to improving the search for a solution has been developed by relying on the inherent structure in the problem as a basis for learning how to characterize solutions. We can compare the decision tree as a data structure to the principle underlying automaton-based approaches discussed above. The advantage of this approach is that we avoid the risk of incurring an exponential space cost, since we can bound the input to limit the size of the decision tree. Also, the decision tree can be seen as having some of the advantages of BFRs, specifically, an improvement in search cost, but without the penalty of solution loss.

5 Conclusions and Future Work

In this paper we have presented a hybrid approach to supporting interactive constraint satisfaction. The hybrid combines both decision tree learning with constraint satisfaction techniques. Our results demonstrate that machine learning techniques provide a very useful basis for improving the performance of constraint satisfaction techniques for interactive problem solving.

There are some interesting issues to be studied here. In particular, we have seen how the classification accuracy of relatively small decision trees is surprisingly high. This raises the following question: what is special, if anything, about the variables and the values referenced in the decision trees? It would be interesting to study this issue in greater detail.

Finally, while the user model currently used here is quite restrictive, being completely system-driven, we plan on studying ways to relax this restriction so user preferences can also be taken into account when finding an initial solution.

References

[Amilhastre *et al.*, 2002] J. Amilhastre, H. Fargier, and P. Marguis. Consistency restoration and explanations in dynamic csps – application to configuration. *Artificial Intelligence*, 135:199–234, 2002.

[Beck *et al.*, 2004] J.C. Beck, T. Carchrae, E. C. Freuder, and G. Ringwelski. Backtrack-free search for real-time constraint satisfaction. In *CP*, LNCS 3258, pages 92–106, 2004.

[Dechter and Pearl, 1987] R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34(1):1–38, 1987.

[Freuder, 1982] E.C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, 1982.

[Freuder, 1990] E.C. Freuder. Complexity of k-tree-structured constraint satisfaction problems. In *Proceedings of AAAI-90*, pages 4–9, 1990.

[Freuder, 1991] E.C. Freuder. Eliminating interchangeable values in constraint satisfaction problems. In *Proceedings of the AAAI*, pages 227–233, 1991.

[O'Sullivan *et al.*, 2004] B. O'Sullivan, A. Ferguson, and E.C. Freuder. Boosting constraint satisfaction using decision trees. In *Proceedings of ICTAI-2004*, pages 646–651, 2004.

[Pu and Faltings, 2004] P. Pu and B. Faltings. Decision tradeoff using example-critiquing and constraint programming. *Constraints*, 9(4):289–310, 2004.

[Quinlan, 1986] J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[Schiex and Verfaillie, 1994] T. Schiex and G. Verfaillie. Nogood recording for static and dynamic constraint satisfaction probles. *Journal on Artificial Intelligence Tools*, 3(2):187–207, 1994.

[Utgoff *et al.*, 1997] P.E. Utgoff, N.C. Berkman, and J.A. Clouse. Decision tree induction based on efficient tree restructuring. *Machine Learning*, 29:5–44, 1997.

Linear Functions for Interactive Configuration Using Join Matching and CSP Tree Decomposition

Sathiamoorthy Subbarayan, Henrik Reif Andersen
Department of Innovation, IT University of Copenhagen
Denmark

Abstract

Quick responses are required for interactive configuration. For this an ideal interactive configurator needs to provide the functionalities required for interactive configuration with at most linear time complexity. In this paper, we present such a data structure called *Join Matched CSP* (JMCSP). When a JMCSP is used to represent a configuration problem, the functionalities required for interactive configuration can be obtained with linear time complexity.

Unlike the tree-of-BDDs [Subbarayan, 2005], the JMCSPs while taking advantage of tree-decomposition also provide linear configuration functions. Although obtaining a JMCSP is exponential in the tree-width of the input configuration problem, due to tree-like hierarchical nature of configuration problems, this is typically feasible. We present the JMCSP compilation process along with the linear interactive configuration functions on it. The linear functionalities provided by the JMCSPs include computation of *all minimum explanations*.

1 Introduction

The complexity of made-to-order products keeps increasing. Examples of such products include personal computers, bikes, and power-backup systems. Such products will be represented in the form of a product model. A product model will list the parameters (variables) defining the product, their possible values and the rules by which those parameter values can be chosen. A product model implicitly represents all valid configurations of a product, and it can be viewed as a Constraint Satisfaction Problem (CSP), where the solutions to the CSP are equivalent to valid configurations of the corresponding product model.

The increase in the complexity of made-to-order products rises the need for efficient decision support systems to configure a product based on the requirements posed by the customer. Such decision support systems are called configurators. Configurators read a product model which represents all valid configurations of the product and guides the user in choosing one among the valid configurations as close as possible to his requirements. An interactive configurator takes a product model as input and interactively helps the user to choose his preferred values for the parameters in the product model, one-by-one.

The interactive configurator needs to be complete and backtrack-free. Complete means that all valid configurations need to be configurable using the configurator. Backtrack-free means that the configurator should not allow the user to choose a value for a parameter in the product model which would eventually lead him to no valid configurations. To ensure backtrack-freeness, the interactive configurator needs to prune away values from the possible values of parameters in the product model as and when those values will not lead to any valid configuration. In addition, the interactive configurator needs to give responses in a short period of time. Examples for commercial interactive configurators include Configit Developer [Configit-Software, 2005] and Array Configurator [Array-Technology, 2005].

The Binary Decision Diagram (BDD) [Bryant, 1986] based symbolic CSP compilation technique [Hadzic *et al.*, 2004; Subbarayan *et al.*, 2004] can be used to compile all solutions of a configuration problem into a single (monolithic) BDD. Once a BDD is obtained, the functions required for interactive configuration can be efficiently implemented. The problem with such approaches is that they do not exploit the fact that configuration problems are specified in hierarchies. Due to this the BDD obtained after compilation could be unnecessarily large. Such hierarchies are closer to trees in shape. The tree-of-BDDs approach: a combination of the BDD-based compilation technique and a CSP decomposition technique for efficient compilation of all solutions was presented in [Subbarayan, 2005]. The tree-of-BDDs scheme exploited the tree-like nature of configuration problems. The advantage of monolithic-BDD is that it can provide linear functions for interactive configuration. The advantage of tree-of-BDDs is that they need very small space to store, when compared with that of the monolithic-BDD. But, at the cost of having costly functions for interactive configuration. The results in [Subbarayan, 2005] have shown that, for functionalities like *minimum explanation generation*, the tree-of-BDDs can take significantly long time to respond. Hence, we would like to have a compilation scheme that takes advantage of the tree-decomposition techniques and at the same time provide linear interactive configuration functions.

Towards this we introduce the notion of *Join Matching* in

tree-structured CSP instances. Given a configuration problem, we can use tree-decomposition techniques to obtain an equivalent tree-structured CSP. Then, we can easily obtain a *join tree* for the tree-structured CSP. By performing Join Matching on the join tree, we obtain a data structure called *Join Matched CSP* (JMCSP). The size of JMCSP is exponential in the tree-width of the input configuration problem. As the configuration problems typically have very low tree-width, this should be practically feasible. We also present the linear functions required for interactive configuration using JMCSPs.

This is a first step towards join matching in other configuration-problem compilation schemes, using the compression data structures like BDDs, DNNF [Darwiche, 2001], automata [Amilhastre *et al.*, 2002; Fargier and Vilarem, 2004], and cartesian product representation [Madsen, 2003]. That might lead to linear functions for interactive configuration with additional reduction in space than just using JMCSPs. The other potential applications of join matching include: *the general constraint propagation techniques, model-based diagnosis, and database systems*.

In Section 2, the basic definitions are given. In Section 3 the CSP tree-decomposition techniques are discussed. The Section 4 describes the join matching process for compiling JMCSPs. The following section presents the interactive configurator algorithm. The Section 6 describes the linear functions for interactive configuration using the JMCSPs. Discussion on future work and related work, followed by concluding remarks, finish this paper.

2 Background

In this section we give the necessary background.

Let X be a set of variables $\{x_1, x_2, \ldots, x_n\}$ and D be the set $\{D_1, D_2, \ldots, D_n\}$, where D_i is the domain of values for variable x_i.

Definition A *relation* R over the variables in M, $M \subseteq X$, is a set of allowed combinations of values (tuples) for the variables in M. Let $M = \{x_{m_1}, x_{m_2}, \ldots, x_{m_k}\}$, then $R \subseteq (D_{m_1} \times D_{m_2} \times \ldots D_{m_k})$. R restricts the ways in which the variables in M could be assigned values.

Definition A *constraint satisfaction problem instance* CSP is a triplet (X, D, C), where $C = \{c_1, c_2, \ldots, c_m\}$ is a set of constraints. Each constraint, c_i, is a pair (S_i, R_i), where $S_i \subseteq X$ is the scope of the constraint and R_i is a relation over the variables in S_i.

We assume that the variables whenever grouped in a set are ordered in a fixed sequence. The same ordering is assumed on any set of the values of variables and the pairs with variables in them.

Definition An *assignment* for a variable x_i is a pair (x_i, v), where $x_i \in X$ and $v \in D_i$. The assignment (x_i, v) binds the value of x_i to v. A partial assignment PA is a set of assignments for all the variables in Y, where $Y \subseteq X$. A partial assignment is complete when $Y = X$.

The notation $\text{PA}_{|xs}$, where xs is a set of variables, means the restriction of the elements in PA to the variables in xs. Similarly, $R_{|xs}$, where R is a relation, means the restriction of the tuples in R to the variables in xs.

Definition Let the set of variables assigned values by a PA be $var(\text{PA}) = \{x_i | \exists x_i.(x_i, v) \in \text{PA}\}$. Let the tuple of values assigned by a PA be $val(\text{PA}) = (v_{l_1}, \ldots, v_{l_j})$ when $var(\text{PA}) = \{x_{l_1}, \ldots, x_{l_j}\}$ and $(x_{l_n}, v_{l_n}) \in \text{PA}$. A partial assignment PA satisfies a constraint c_i, when $val(\text{PA}_{|var(\text{PA}) \cap S_i}) \in R_{i|var(PA) \cap S_i}$.

Definition A *complete assignment* CA is a *solution* S for the CSP when CA satisfies all the constraints in C.

Given a CSP instance, the set of all complete assignments is $\text{CAS} \equiv D_1 \times D_2 \times \ldots \times D_n$. Let SOL denote the set of all solutions of the CSP. Then, $\text{SOL} \subseteq \text{CAS}$.

A *configurator* is a tool which helps the user in selecting his preferred product, which needs to be valid according to a given product specification. An *interactive configurator* is a configurator which interacts with the user as and when a choice is made by the user. After each and every choice selection by the user the interactive configurator shows a list of unselected options and the valid choices for each of them. The interactive configurator only shows the list of valid choices to the user. This prevents the user from selecting a choice, which along with the previous choices made by the user, if any, will result in no valid product according to the specification. The configurator, hence, automatically hides the *invalid choices* from the user. The invalid choices will still be visible to the user, but with a tag that they are inconsistent with the current partial assignment. When the current partial assignment is extended by the user, some of the choices might be implied for consistency. Such choices are automatically selected by the configurator and they are called *implied choices*.

A configuration problem can be modelled as a CSP instance, in which each available option will be represented by a variable and the choices available for each option will form the domain of the corresponding variable. Each rule in the product specification can be represented by a corresponding constraint in the CSP instance. The CAS and SOL of the CSP instance will denote the all possible-configurations and all possible-valid-configurations of the corresponding configuration problem. Hereafter, the terms CSP, CAS, and SOL may be used directly in place of the corresponding configuration terms.

Let us assume that the SOL of a configuration problem can be obtained. Given SOL, the three functionalities required for interactive configuration are: *Display*, *Propagate*, and *Explain*.

Definition Display is the function, which given a CSP, a subset of its solutions space SOL' $\subseteq$ SOL, lists X, the options in CSP and CD_i, the available valid choices, for each option $x_i \in X$, where $CD_i = \{v | (x_i, v) \in S, S \in \text{SOL'}\}$.

CD_i is the current valid domain for the variable x_i. The display function is required to list the available choices to the user.

Definition Propagate is the function, which given a CSP, a subset of its solutions space SOL' $\subseteq$ SOL, and (x_i, v), where $v \in CD_i$, restricts SOL' to $\{S | (x_i, v) \in S, S \in \text{SOL'}\}$.

By the definition of interactive configurator, the propagation function is necessary. Propagate could also be written as restricting SOL' to SOL'$_{|(x_i,v)}$. Sometimes the restriction might involve a set of assignments, which is equivalent to making each assignment in the set one by one.

Definition A choice (x_i, v_i) is an *implied choice*, when $(x_i, v_i) \notin$PA and $(\text{SOL}_{|\text{PA}})_{|x_i} = \{v_i\}$. A choice (x_i, v_i) is an *invalid choice*, when $\{v_i\} \notin (\text{SOL}_{|\text{PA}})_{|x_i}$. Let (x_i, v_i) be an implied or invalid choice. Then *Explain*(x_i, v_i) is the process of generating, E, a set of one or more selections made by the user, which implies or invalidates (x_i, v_i). The E is called as an explanation for (x_i, v_i).

An explanation facility is required when the user wants to know why a choice is implied or invalid. Let PA be the current partial assignment that has been made by the user. By the definition of explanation, *Display*$(\text{CSP}, \text{SOL}_{|\text{PA}\setminus\text{E}})$ will list v_i as a choice for the unselected option x_i.

Each selection, (x_i, v), made by the user could be attached a non-negative value as its priority, $P(x_i, v)$, and the explain function can be required to find a minimum explanation.

Definition The *cost* of an explanation is Cost(E) $= \sum_{(x_i,v)\in E} P(x_i, v)$. An explanation E is *minimum*, if there does not exists an explanation E' for (x_i, v_i), such that Cost(E')<Cost(E).

Minimum explanations are useful when different options in a product model have different priorities. For example, in a car configuration problem the main options like engine could be given high priority. Once the user decides on an option of high priority, minimum explanations will try to protect the high priority decision as much as possible.

The complexity of the three functions —display,propagate and explain— are exponential in the size of the input configuration problem.

3 Tree Decomposition of CSPs

The following definitions [Dechter, 2003] will be useful in discussing the tree decomposition techniques for CSPs.

Definition The *constraint graph* (V,E) of a given CSP will contain a node for each constraint in the CSP and an edge between two nodes if their corresponding constraints share at least one variable in their scopes. Each edge will be labelled by the variables that are shared by the scope of the corresponding constraints.

Definition A subset of the edges, ($E' \subseteq E$), in a constraint graph is said to satisfy the *connectedness* property, if for any variable v shared by any two constraints, there exists a path between the corresponding nodes in the constraint graph, using only the edges in E' with v in their labels.

Definition A *join graph* (V,E_j) of a constraint graph contains all the nodes in the constraint graph, but the set of edges in the join graph, E_j, is a subset of the edges in the constraint graph, $E_j \subseteq E$, such that the connectedness property is satisfied. If the join graph does not have any cycles, then it is called a *join tree*.

A constraint graph of a CSP has a join tree if its *maximum spanning tree*, when the edges are weighted by the number of shared variables, satisfies the connectedness property [Dechter, 2003]. Several tree decomposition techniques [Dechter and Pearl, 1989; Gyssens *et al.*, 1994; Gottlob *et al.*, 2000] have been proposed to convert a CSP C into *C'* that has a join tree.

Definition The *tree width* of a join tree is the maximum number of variables in any of its nodes minus one. The tree width of a CSP is the smallest one among the tree width of all possible join trees.

Finding a tree decomposition of a CSP such that the join tree of the resulting CSP has the minimum tree width is a NP-hard task [Bodlaender, 2005]. Hence, most of the tree decomposition techniques use some form of heuristics. The complexity of creating a join tree of a CSP is exponential in the tree width of the join tree.

All the tree decomposition techniques create clusters of constraints in a CSP, such that all the constraints in the CSP will be in at least one of the clusters. The CSP' obtained after the conjunction of the original constraints in each of the clusters will have a join tree. A solution for CSP or CSP' will also be a solution for the other.

Definition A constraint of a CSP is said to be *minimal* when all the solutions of the constraint can be extended to a solution for the CSP.

A CSP with a join tree is called *acyclic (tree-structured) CSP*. Acyclic CSPs can be efficiently solved. The nice property of an acyclic CSP is that, local consistency between constraints adjacent in its join tree implies minimality of constraints. Minimality of constraints is enough to obtain the efficient functions required for interactive configuration. Since each constraint is minimal, a valid choice for a variable in a constraint will also be a valid choice for the variable in the entire CSP, and the display function will use this to efficiently display valid choices. Given an assignment, the propagate function just needs to reduce all the constraints in which the variable is present, and propagate the effect to other constraints through *semi-joins* [Goodman and Shmueli, 1982].

Although the resulting CSP after decomposition can be exponential in the worst case, for many practical configuration instances the decomposition results in a manageable sized acyclic CSP [Subbarayan, 2005].

4 Join Matched CSPs

Definition Let *C* be an acyclic CSP and its join tree be *(V,E)*, where each element in *V* corresponds to a constraint in *C* and the edges in E are undirected. Then, a *Directed Join Tree* (DJT) is a pair *(V,E')*, where all the edges in *E'* are directed such that, each edge *e*$\in$*E* will be converted into a directed edge *e'*$\in$*E'*, and only one node in *V* will not have any outgoing edge.

Let the conjunction of the constraints c_1 and c_2 be $(c_1 \bowtie c_2)$. Let ca be the common variables in the scope of those constraints.

Definition The *common join values* (CJV) of an edge *e'* between the constraints c_1 and c_2 is defined as $CJV = (c_1 \bowtie c_2)_{|ca}$.

Definition A *common join block* (CJB) is a structure, associated with a directed edge, containing the fields: *cja, left_ptrs, right_ptrs, min_cost, min_ptr*.

The *cja* in a CJB belongs to the corresponding CJV, i.e. *cja* $\in$ CJV. The *left_ptrs* is a list of pointers to the tuples on the source constraint of the corresponding directed edge. The list points to the tuples which when restricted to common variables of the corresponding edge evaluates to *cja*. Similarly, the *right_ptrs* will point to the tuples containing *cja* on the destination constraint. The fields *min_cost* and *min_ptr* will be used by the Explain function and they are described later.

Given an acyclic CSP *C* and its directed join tree, for each edge in the join tree we can obtain the corresponding CJV. Then for each element of CJV, we can obtain the corresponding CJB.

Definition Let *CJBS* of an edge be the set of CJBs, having a CJB for each value in the corresponding common join values (CJV).

Whenever a CJBS is created for a directed edge e from c_1 to c_2, for each tuple in c_1 and c_2, we add a pointer to the corresponding CJB in the CJBS. The lists *left_ptrs* and *right_ptrs* in the corresponding CJBs will also be updated.

Given an edge e, the process of creating CJBS and adding appropriate pointers to the tuples in c_1 and c_2 is called *Join Matching* for the edge e.

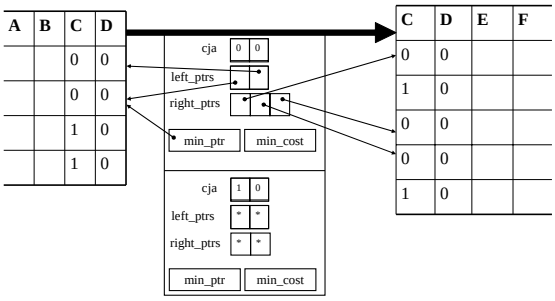


Figure 1: Join Matching for an edge.

In the example of Figure 1, the common attributes are {*C,D*}. There are two elements in the CJV {(0,0),(1,0)}. For the CJB corresponding to *cja* (0,0), the pointers stored in the CJB are marked by appropriate arrows. Note that each tuple will also have a pointer to one of those CJBs, but the figure does not show them explicitly.

Definition Given an acyclic CSP and its directed join tree, the process of join matching for each edge in the join tree results in a data structure called *Join Matched CSP* (JMCSP).

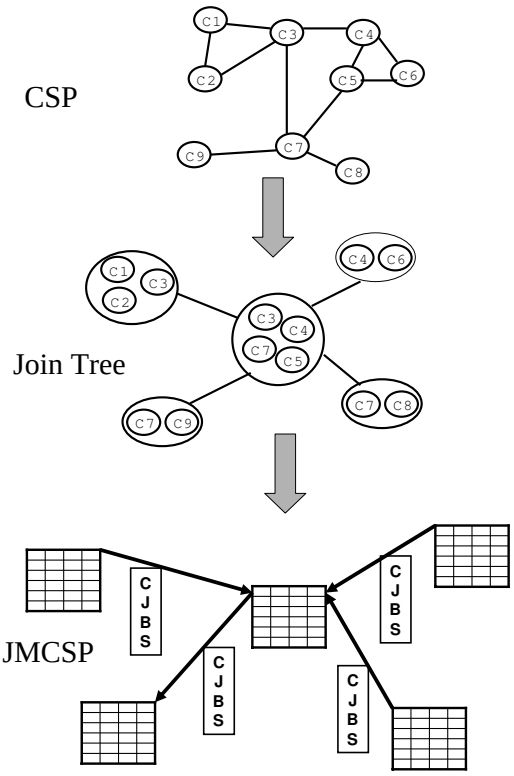


Figure 2: CSP to JMCSP conversion.

Figure 2 shows an example for JMCSP creation for a configuration problem represented by its constraint graph.

After creating a JMCSP, minimality of the constraints in it can be obtained by removing tuples in the constraints, which do not have a counterpart in an adjacent constraint. Iteration of this process will reach a *fix point*, where all the constraints in the JMCSP will be minimal.

The Complexity of Join Tree to JMCSP Compilation

It can observed that the join matching process increases the space of each tuple by a constant. If there are totally t tuples in a join tree, then the size of JMCSP is O(*t*). Thus the size of JMCSP is linear in the size of the join tree.

Let a join tree has n nodes (constraints) and there are l tuples in each one of those nodes. Join matching of an edge can be done by first sorting the two end constraints of the edge on the common attributes (variables). The sorting of the constraints take O(*l log l*) time. Then, an O(*l*) algorithm is enough to obtain the corresponding CJBS as both the end constraints are lexicographically ordered on the join attributes. Hence, the join matching of an edge can be done in O(*l log l*) time. As there are *n-1* edges in the join tree, the join tree to JMCSP conversion process takes O(*n l log l*) time.

5 An Interactive Configurator Using JMCSPs

The algorithm for interactive configuration is given in Figure 3. The COMPILE function takes a configuration problem as input, converts it into an acyclic CSP, and then returns the corresponding JMCSP. The JMCSP represents the solution space (SOL). Since a configuration problem will not change quite often, the COMPILE function need not have to do the compilation every time the interactive configurator is used. SOL could be stored and reused.

```
INTERACTIVECONFIGURATOR(CP)
1  SOL:=COMPILE(CP)
2  SOL':=SOL, PA:={ }
3  while |SOL'| > 1
4    Display(CP, SOL')
5    (x_i,v) := 'User input choice'
6    if (x_i,v)∈ CD_i
7      SOL':=Propagate(CP,SOL',(x_i,v))
8      PA:=PA∪{(x_i,v)}
9    else
10     E:=Explain(CP, SOL', (x_i,v))
11     if 'User prefers (x_i,v)'
12       PA:=(PA\ E)∪{(x_i,v)}
13       SOL':=SOL_|PA
14 return PA
```

Figure 3: An Interactive Configurator Using JMCSPs.

6 Configuration Functions on JMCSPs

6.1 The Propagate Function

Given a JMCSP and an assignment (x,v), we just have to restrict (mark appropriately) the tuples of one among the constraints having the variable x and propagate the changes to the adjacent constraints in the join tree. Using the CJBS lists, we can obtain the list of non-compliant tuples in the adjacent constraints and mark them as removed. During the process, if all the tuples corresponding to any CJB are removed, then the corresponding CJB is marked as removed and the effect is propagated. The Propagate function should not consider the direction of the edges in the join tree. Due to the properties of acyclic CSPs, such a propagation is enough to maintain minimality after unary assignments. The time complexity of Propagate is linear in the size of the JMCSP.

6.2 The Display Function

Given a JMCSP, with tuples in it marked as removed or not, the Display function just has to access all the tuples once and obtain the allowed values for the unassigned variables. This function also has a linear time complexity.

6.3 The Explain Function

Given a JMCSP, a partial assignment PA, a cost-function for each assignment in PA, and an assignment (x_i,v_i) for which explanation is required, *all* minimum explanations can be obtained as follows.

Definition A *topological ordering* of nodes (constraints) in a JMCSP, is an ordering such that, if c_1 and c_2 are two nodes in the JMCSP, then c_1 precedes c_2 in the ordering, if c_2 can be reached from c_1.

A topological ordering can be obtained in linear time during the JMCSP creation process.

Definition The *valuation* is a function which given a PA and a corresponding JMCSP, maps a tuple in the JMCSP to a positive integer value or ∞. The valuation function is defined as follows:

1. Each element in the tuple corresponding to an unassigned variable will contribute nothing to the valuation.
2. Each element in the tuple that violates an assignment in PA will contribute the value given by the cost-function, for that assignment.
3. The cost for violation of (x_i,v_i) is ∞.
4. If the tuple contains a pointers to a CJB of an incoming directed edge, then the *min_cost* field in the CJB will contribute towards the valuation.

After a valuation of a tuple is obtained, it is compared with the *min_cost* value in the corresponding CJBs of its outgoing edges, if any. If the existing *min_cost* is larger than the valuation of the tuple then, the *min_cost* field is assigned the valuation of the tuple and the corresponding *min_ptr* is updated to point to the tuple.

When the tuples of the constraint (node) without any outgoing edge are valuated, a tuple with minimum valuation in the constraint will be obtained. Recall that there can be only one such node.

Given these steps, a minimum explanation can be obtained as follows:

1. Assign ∞ to all *min_cost* fields in the JMCSP.
2. Following the topological ordering, select constraints and obtain valuation for their tuples.
3. All the *min_ptr* pointed tuples and the cheapest tuple in the last node now gives a minimum explanation. The assignments in PA that are violated by those tuples is the minimum explanation.

Note that all minimum explanations can be obtained by remembering all the *min_cost* valuated tuples in each node. Even with that facility, the Explain function will just have a linear time complexity.

7 Future Work

The tree-of-BDDs [Subbarayan, 2005] scheme has two types of space reduction. Potentially exponential space reduction due to tree-decomposition and another potential exponential space reduction due to BDD-based representation of each constraint. The JMCSPs just takes advantage of tree-decomposition and provides linear functions. The next step will be to adapt the join matching process for tree-of-BDDs. This seems plausible although not the generation of all minimum explanations. But, generation of one minimum explanation seems easy. Further research is required in this direction.

The join matching process has potential applications in general constraint propagation, model-based diagnosis, and database systems. The join matching might reduce the complexity of some of the functions in these applications.

8 Related Work

In [Fattah and Dechter, 1995; Stumptner and Wotawa, 2001], the authors have presented techniques for generating *minimal diagnosis* in model-based diagnosis systems. The problem of minimal diagnosis is very similar to our problem of generating minimum explanations. But they use sorting operations on constraints in the diagnosis process, and this increases the complexity of the operations by a logarithmic factor. Such a logarithmic factor might influence a lot, since the constraints in the real-world instances could have several thousand tuples in a single constraint, even before decomposition. After tree decomposition the number of tuples in the constraints of the resulting acyclic CSP is normally more than the number of tuples in the constraints before decomposition. Hence, it will take a significant amount of time to sort those constraints while generating explanations. For example, the *Renault car configuration* instance used in [Amilhastre *et al.*, 2002] has around 40,000 tuples in a constraint, even before decomposition. Also, in their complexity analysis a linear factor is *suppressed* and hence their minimal diagnosis algorithms will demand more time.

The JMCSPs, whose data structures remains relatively *static* during the explanation generation process, can hence be used in generating minimal diagnosis without any complex steps like sorting used in [Fattah and Dechter, 1995; Stumptner and Wotawa, 2001].

In [Madsen, 2003], the author uses some preprocessing techniques along with tree-decomposition and *cartesian product representation* for interactive configuration. The author was able to provide linear propagation functions but not explanations.

9 Conclusion

The JMCSPs, a data structure with linear configuration functions was presented. Experiments on real-life instances need to be done to empirically test the usefulness of JMCSPs. The join matching technique in JMCSPs can be combined with the compression capability of BDDs or automata [Amilhastre *et al.*, 2002] such that we get an additional decrease in space, while having linear configuration functions.

Acknowledgements

Special thanks to Tarik Hadzic for his comments on a previous version of this paper. Erik van der Meer has independently developed a dynamic version of Tree-of-BDDs in his PhD thesis and discussions with him were helpful.

References

[Amilhastre *et al.*, 2002] J. Amilhastre, H. Fargier, and P. Marquis. Consistency restoration and explanations in dynamic CSPs-application to configuration. *Artificial Intelligence*, 1-2:199–234, 2002.

[Array-Technology, 2005] Array-Technology. `http://www.array.dk/`, 2005.

[Bodlaender, 2005] Hans L. Bodlaender. Discovering treewidth. In *Proceedings of SOFSEM*, pages 1–16, 2005. Springer LNCS.

[Bryant, 1986] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 8:677–691, 1986.

[Configit-Software, 2005] Configit-Software. `http://www.configit-software.com`, 2005.

[Darwiche, 2001] Adnan Darwiche. Decomposable negation normal form. *Journal of the ACM*, 48(4):608–647, 2001.

[Dechter and Pearl, 1989] R. Dechter and J. Pearl. Tree-clustering schemes for constraint-processing. *Artificial Intelligence*, 38(3):353–366, 1989.

[Dechter, 2003] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[Fargier and Vilarem, 2004] H. Fargier and M-C. Vilarem. Compiling CSPs into tree-driven automata for interactive solving. *Constraints*, 9(4):263–287, 2004.

[Fattah and Dechter, 1995] Yousri El Fattah and Rina Dechter. Diagnosing tree-decomposable circuits. In *Proceedings of IJCAI*, pages 1742–1749, 1995.

[Goodman and Shmueli, 1982] Nathan Goodman and Oded Shmueli. The tree property is fundamental for query processing. In *Proceedings of PODS*, pages 40–48, 1982.

[Gottlob *et al.*, 2000] Georg Gottlob, Nicola Leone, and Francesco Scarcello. A comparison of structural CSP decomposition methods. *Artificial. Intelligence*, 124(2):243–282, 2000.

[Gyssens *et al.*, 1994] Marc Gyssens, Peter G. Jeavons, and David A. Cohen. Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence*, 66(1):57–89, 1994.

[Hadzic *et al.*, 2004] T. Hadzic, S. Subbarayan, R. M. Jensen, H. R. Andersen, J. Møller, and H. Hulgaard. Fast backtrack-free product configuration using a precompiled solution space representation. In *Proceedings of PETO conference*, pages 131–138, 2004.

[Madsen, 2003] J. N. Madsen. Methods for interactive constraint satisfaction. Master's thesis, Department of Computer Science, University of Copenhagen, 2003.

[Stumptner and Wotawa, 2001] Markus Stumptner and Franz Wotawa. Diagnosing tree-structured systems. *Artificial Intelligence*, 127(1):1–29, 2001.

[Subbarayan *et al.*, 2004] S. Subbarayan, R. M. Jensen, T. Hadzic, H. R. Andersen, H. Hulgaard, and J. Møller. Comparing two implementations of a complete and backtrack-free interactive configurator. In *CP'04 CSPIA Workshop*, pages 97–111, 2004.

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

Debugging User Interface Descriptions of Knowledge-based Recommenders

Alexander Felfernig, Shchekotykhin Kostyantyn
University Klagenfurt, Computer Science and Manufacturing
Universitätsstrasse 65-67, A-9020 Klagenfurt, Austria
alexander.felfernig@uni-klu.ac.at, kostyantyn.shchekotykhin@ifit.uni-klu.ac.at

Abstract

Finite state automata are a suitable representation formalism for describing the intended behaviour of the user interface of a knowledge-based recommender application. Such interfaces have a finite number of states, where transitions are triggered by a set of user requirements. Unfortunately, faulty models of the user interface can be easily defined by knowledge engineers and no automated support for debugging such definitions is available. This paper presents an approach to automated debugging of faulty user interface definitions of knowledge-based recommenders which is highly relevant for increasing the productivity in user interface implementation. The goal of the paper is to show how concepts from model-based diagnosis can be applied in order to identify minimal sources of inconsistencies in user interface designs. The presented approach has been implemented as a prototype on the basis of a commercially available knowledge-based recommender environment.

1 Introduction

Knowledge acquisition and maintenance as a collaborative process between technical and domain experts is still a time consuming task where time savings related to fault identification play an important role [Felfernig *et al.*, 2004]. In this paper we focus on a situation where knowledge engineers develop and maintain a model of the user interface of a knowledge-based recommender. Such interactive applications can be described by a finite number of states, where state transitions are triggered by user requirements representing a set of variable settings which serve as input for e.g. the constriant solver or configurator of the recommender environment. Finite state automata [Hopcroft and Ullman, 1979] are a suitable representation formalism for expressing the expected behavior of a recommender user interface [Bass and Coutaz, 1991; Wasserman, 1985]. Figure 1 depicts a simple example for the intended behaviour of a user interface of a financial services recommender application. Typically, in an interactive recommendation process customers make decisions by specifying values (requirements) for a subset of a given set of variables. Depending on the answers given by a customer, a corresponding state in the automaton is reached, e.g. an expert ($kl = expert$) who doesn't want to be guided through a financial services advisory process ($aw = no$) is directly forwarded to the state q_3, where a direct product search (search based on product parameters) can be performed, i.e. different subsets of relevant variables are determined by paths of the automaton. Note that the design in Figure 1 is faulty: an expert ($kl = expert$) who wants to be guided by a financial advisory process ($aw = yes$) and is interested in long-term investments ($id = longterm$) and doesn't have any available funds ($av = no$) comes to a standstill at the input of availability since the conditions $\{c_2, c_9\}$ and $\{c_2, c_{11}\}$ are contradictory. Such situations occur very often when designing interfaces for knowledge-based recommenders. Consequently, additional support is needed for the design of the recommender user interface, basically in the form of automated identification of faulty definitions. In the following we show how concepts from model-based diagnosis (MBD) can be applied in order to identify a minimal set of changes which allow consistency recovery in finite state representations of a recommender user interface. The concepts presented in this paper have been prototypically implemented within the scope of the Koba4MS project[1], where finite state models are used to specify the intended behaviour of a user interface. Such a finite state specification can be automatically translated into a corresponding graphical recommender interface.

The remainder of the paper is organized as follows. In Section 2 we introduce the concepts of predicate-based finite state automata ($\mathcal{PFSA}$), a formalism for modeling the navigational behaviour of user interfaces. In Section 3 we provide a formal basis for the diagnosis of a $\mathcal{PFSA}$. Section 4 contains an evaluation of the presented concepts. The paper closes with a discussion on related work.

2 Predicate-based Automata

Finite state automata are a means to define the intended navigational behaviour of interactive recommender applications. In contrast to conventional finite automata formalizations (see e.g. [Hopcroft and Ullman, 1979]) we introduce the notion of predicate-based finite state automata ($\mathcal{PFSA}$) [v.Noord

[1]Knowledge-based Advisors for Marketing and Sales is a project funded by the Austrian Research Fund under the agreement number FFF-808479 and the European Union (20-REG-1025/12-2003).

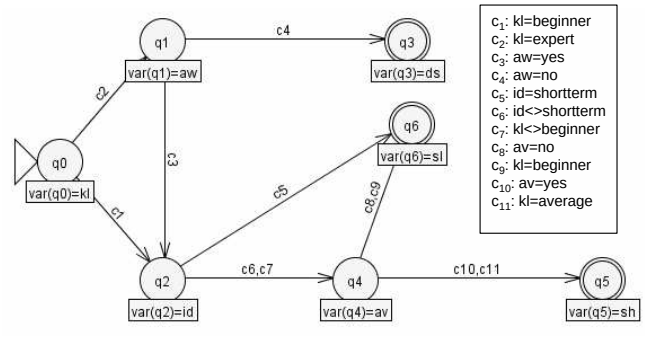


Figure 1: Example (faulty) $\mathcal{PFSA}$. $\{c_1, c_2, ..., c_{11}\}$ are transition conditions between states q_i, $var(q_i)$ represents the variable which is instantiated by the user in the state q_i.

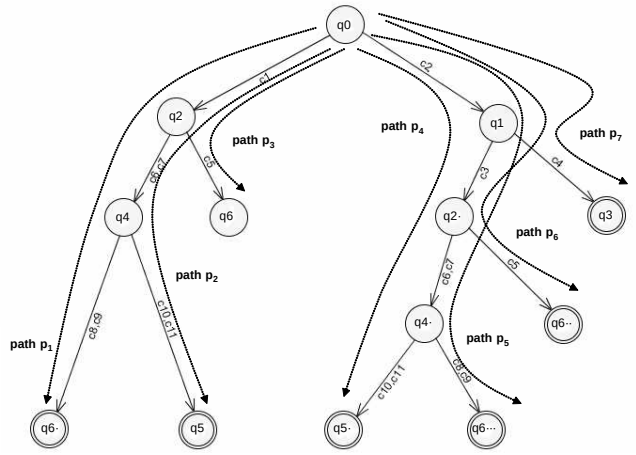


Figure 2: Reachability tree of $\mathcal{PFSA}$ which is the expansion of the $\mathcal{PFSA}$ shown in Figure 2. $\{p_1, p_2, ..., p_7\}$ are the possible (not necessarily accessible) paths of the $\mathcal{PFSA}$.

and Gerdemann, 2001] which is a more natural and compact approach to define transitions between different interaction phases (transitions are defined in terms of domain restrictions). It is more intuitive to define an interaction path restricted by the condition kl <> $beginner$ than to define a specific transition for each possible value from the variable domain of kl ($beginner$, $average$, $expert$), i.e. a potentially large set of transitions is replaced by a single transition. For the following descussions we introduce the notion of a predicate-based finite state automaton ($\mathcal{PFSA}$), where we restrict our discussions to the case of acyclic automata.

Definition 1 ($\mathcal{PFSA}$): we define a Predicate-based Finite State Automaton (recognizer) ($\mathcal{PFSA}$) to be a 6-tuple (Q, Σ, Π, E, S, F), where

- Q = {q_1, q_2, ..., q_m} is a finite set of states, where $var(q_i)$ = x_i is a finite domain variable assigned to q_i, $prec(q_i)$ = {ϕ_1, ϕ_2, ..., ϕ_m} is the set of preconditions of q_i (ϕ_k = {c_m, c_n, ..., c_o} $\subseteq \Pi$), $postc(q_i)$ = {ψ_1, ψ_2, ..., ψ_n} is the set of postconditions of q_i (ψ_l = {c_m, c_n, ..., c_o} $\subseteq \Pi$), and $dom(x_i)$ = {x_i=d_{i1}, x_i=d_{i2}, ..., x_i=d_{ik} } denotes the set of possible assignments of x_i, i.e. the domain of x_i.
- Σ = {$x_i = d_{ij} \mid x_i = var(q_i)$, $(x_i = d_{ij}) \in dom(x_i)$} is a finite set of variable assignments (input symbols), the input alphabet.
- Π = {c_1, c_2, ..., c_p} is a set of constraints (transition conditions) restricting the set of words accepted by the $\mathcal{PFSA}$.
- E is a finite set of transitions $\subseteq Q \times \Pi \times Q$.
- $S \subseteq Q$ is a set of start states.
- $F \subseteq Q$ is a set of final states. □

The set of preconditions for a state q_i, i.e. prec(q_i) = {ϕ_1, ϕ_2, ..., ϕ_n} represents a disjunction of invariants in the state q_i. These invariants can be automatically derived from the reachability tree of a $\mathcal{PFSA}$ which is a common approach to study properties of finite state models. See e.g. Figure 2 which represents the reachability tree for the $\mathcal{PFSA}$ depicted in Figure 1. The state q_2 is accessed twice in the reachability tree, the preconditions of q_2 (i.e. $prec(q_2)$) can be directly derived from the transition conditions of the paths leading to q_2, i.e. {{c_1}, {c_2,c_3}}. Similarly, $postc(q_i)$ represents the set of possible postconditions of the state q_i which as well can be derived from the reachability tree.

The following is an example for a $\mathcal{PFSA}$ describing the behaviour of a financial services advisory dialog which conforms to the graphical definition of Figure 1. The $\mathcal{PFSA}$ defines possible navigation sequences in a financial services adivory dialog. States in the automata represent questions which are posed to customers. Depending on the answers given by a customer, a subsequent state is selected where additional questions can be posed to the customer. An advisory process is completed when a final state is reached. The set of input sequences leading to a final state is also denoted as the language accepted by the $\mathcal{PFSA}$.

Example 1 ($\mathcal{PFSA}$): {Q = {q_0, q_1, q_2, q_3, q_4, q_5, q_6}.

$var(q_0)$ = *kl*. /* knowledge level of the customer */
$var(q_1)$ = *aw*. /* advisory wanted */
$var(q_2)$ = *id*. /* duration of investment */
$var(q_3)$ = *ds*. /* direct specification */
$var(q_4)$ = *av*. /* availability of financial resources*/
$var(q_5)$ = *sh*. /* high risk products */
$var(q_6)$ = *sl*. /* low risk products */
dom(*kl*) = {*kl=beginner, kl=average, kl=expert*}.
dom(*aw*) = {*aw=yes, aw=no*}.
dom(*id*) = {*id=shortterm, id=mediumterm, id=longterm*}.
dom(*ds*) = {*ds=savings, ds=bonds,*
ds=stockfunds, ds=singleshares}.
dom(*av*) = {*av=yes, av=no*}.
dom(*sh*) = {*sh=stockfunds, sh=singleshares*}.
dom(*sl*) = {*sl=savings, sl=bonds*}.
$prec(q_0)$ = {{$true$}}. $prec(q_1)$ = {{c_2}}.
$prec(q_2)$ = {{c_1}, {c_2,c_3}}. $prec(q_3)$ = {{c_2, c_4}}.
$prec(q_4)$ = {{c_1, c_6, c_7}, {c_2, c_3, c_6, c_7}}.
$prec(q_5)$ = {{$c_1, c_6, c_7, c_{10}, c_{11}$}, {$c_2, c_3, c_6, c_7, c_{10}, c_{11}$}}.

$prec(q_6)$ = {{c_1, c_5}, {c_2, c_3, c_5}, {c_1, c_6, c_7, c_8, c_9}, {$c_2, c_3, c_6, c_7, c_8, c_9$}}.
Σ= {*kl=beginner, kl=average, kl=expert, aw=yes, aw=no, ..., sl=savings, sl=bonds*}.
$postc(q_0)$ = {{c_2, c_4}, {c_2, c_3, c_5}, {c_2, c_3, c_6, c_7, c_8, c_9}, {c_2, c_3, c_6, c_7, c_{10}, c_{11}}, {c_1, c_5}, {c_1, c_6, c_7, c_8, c_9}, {c_1, c_6, c_7, c_{10}, c_{11}}}.
$postc(q_1)$ = {{c_4}, {c_3, c_5}, {c_3, c_6, c_7, c_8, c_9 }, {c_3, c_6, c_7, c_{10}, c_{11}}}.
$postc(q_2)$ = {{c_5}, {c_6, c_7, c_8, c_9}, {c_6, c_7, c_{10}, c_{11}}}.
$postc(q_3)$ = {{$true$}}. $postc(q_4)$ = {{c_8, c_9}, {c_{10}, c_{11}}}.
$postc(q_5)$ = {{$true$}}. $postc(q_6)$ = {{$true$}}.
Π = {c_1, c_2, ..., c_{11}}.
E= {(q_0, {c_2}, q_1), (q_0, {c_1}, q_2), (q_1, {c_4}, q_3), (q_1, {c_3}, q_2), (q_2, {c_6, c_7}, q_4), (q_2, {c_5}, q_6), (q_4,{c_8, c_9}, q_6), (q_4, {c_{10}, c_{11}}, q_5)}.
S= {q_0}. F= {q_3, q_5, q_6}.} □

A word $w \in \Sigma^*$ (i.e. a sequence of user inputs) is accepted by a $\mathcal{PFSA}$ if there is an accepting run of w in $\mathcal{PFSA}$. The set of accepted words can be defined as follows.

Definition 2 ($\mathcal{PFSA}$-accepted language): The relation $E' \subseteq Q \times \Sigma^* \times Q$ specifying the set of words $w \in \Sigma^*$ accepted by $\mathcal{PFSA}$ is defined inductively as follows:

- for all (q_i, π, q_j) $\in E$ ($q_i \in S$) and for all $\sigma \in dom(var(q_i))$: if consistent ($\pi \cup \{\sigma\}$) then (q_i, [σ], q_j) $\in E'$.
- for all (q_i, [σ_1, σ_2, ..., σ_n], q_j) $\in E'$, (q_j, π, q_k) $\in E$, $\sigma_{n+1} \in dom(var(q_j))$: if $\exists\ \psi \in postc(q_j)$ s.t. consistent($\psi \cup \{\sigma_1, \sigma_2, ..., \sigma_n\} \cup \{\sigma_{n+1}\}$) then (q_i, [σ_1, σ_2, ..., σ_n, σ_{n+1}], q_k) $\in E'$.

A $\mathcal{PFSA}$-accepted language is defined by the set of accepted words $w \in \Sigma^* : \{w \mid q_0 \in S, q_f \in F, (q_0, [w], q_f) \in E'\}$. □

When developing and maintaining user interfaces, mechanisms need to be provided which support the effective identification of violations of well-formedness properties, e.g. if an input sequence reaches state q_i, there must be at least one extension of this input sequence to a final state. Path expressions form the basis for expressing such well-formedness properties on a $\mathcal{PFSA}$.

Definition 3 (consistent path): Let p = [(q_1,C_1,q_2), (q_2,C_2,q_3), ..., (q_{i-1},C_{i-1},q_i)] ((q_i,C_i,q_j) $\in E$) be a path from a state $q_1 \in S$ to a state $q_i \in Q$. p is consistent ($consistent(p)$) *iff* $\bigcup C_j$ is satisfiable. □

Based on the definition of a consistent path we can introduce the following well-formedness rules which specifiy structural properties of a $\mathcal{PFSA}$. For each consistent path in a $\mathcal{PFSA}$ leading to a state q_i there must exist a corresponding *direct* postcondition (i.e. q_i, C_i, q_j) propagating the path (i.e. each consistent path must be *extensible*).

Definition 4 (extensible path): Let p = [(q_1,C_1,q_2), (q_2,C_2,q_3), ..., (q_{i-1},C_{i-1},q_i)] be a path from a state $q_1 \in S$ to a state $q_i \in Q$ - F. p is extensible ($extensible(p)$) *iff* $\exists$ (q_i,C_i,q_{i+1}) : $C_1 \cup C_2 \cup ... \cup C_{i-1} \cup C_i$ is satisfiable. □

Each state q_i is a decision point for the next state determined by the $\mathcal{PFSA}$. This selection strictly depends on the definition of the *direct* postconditions for q_i, where each postcondition has to be unique for determining the subsequent state. A state q_i is *deterministic* if each of its postconditions is unique for determining subsequent states.

Definition 5 (deterministic state): Let p = [(q_1,C_1,q_2), (q_2,C_2,q_3), ..., (q_{i-1},C_{i-1},q_i)] be a path from a state $q_1 \in S$ to a state $q_i \in Q$ - F. A state (q_i) is $deterministic(q_i)$ *iff* $\forall$(q_i, C_{i1}, q_j), (q_i, C_{i2}, q_k) $\in E : C_1 \cup C_2 \cup ... \cup C_{i-1} \cup C_{i1} \cup C_{i2}$ is contradictory ($C_{i1} \neq C_{i2}$). □

Each transition should be *accessible*, i.e. for each transition condition there exists at least one corresponding path.

Definition 6 (accessible transition): A transition t = (q_i,C_i,q_{i+1}) (postcondition of state q_i) is accessible ($accessible(t)$) *iff* there exists a path p = [(q_1,C_1,q_2), (q_2,C_2,q_3), ..., (q_{i-1},C_{i-1},q_i)] ($q_1 \in S$): $C_1 \cup C_2 \cup ... \cup C_{i-1} \cup C_i$ is satisfiable. □

A $\mathcal{PFSA}$ is well-formed, if the given set of well-formedness rules is fulfilled.

Definition 7 (well-formed $\mathcal{PFSA}$): A $\mathcal{PFSA}$ is well-formed *iff*

- each consistent path p = [(q_1,C_1,q_2), (q_2,C_2,q_3), ..., (q_{i-1},C_{i-1},q_i)] ($q_1 \in S$, $q_i \in Q$ - F) is extensible to a consistent path p' = {(q_1,C_1,q_2), (q_2,C_2,q_3), ..., (q_{i-1},C_{i-1},q_i), (q_i,C_i,q_j)}.
- $\forall q_k \in Q$: $deterministic(q_k)$.
- $\forall t$ = (q_k,C_k,q_l) $\in E$: $accessible(t)$. □

3 $\mathcal{PFSA}$-Debugging

In the case of a not well-formed $\mathcal{PFSA}$ we have to (automatically) identify a minimal set of transition conditions in the $\mathcal{PFSA}$ responsible for the wrong behaviour. For this purpose we apply model-based diagnosis (MBD) [Reiter, 1987] by introducing the notion of a $\mathcal{PFSA}$ Diagnosis Problem and a corresponding $\mathcal{PFSA}$ Diagnosis. The MBD approach starts with the description of a system (SD) which is in our case the structural description and the intended behaviour of a $\mathcal{PFSA}$ (Definitions 4-6). If the actual behaviour conflicts with the intended system behaviour, the diagnosis task is it to determine those system components (transition conditions) which, when assumed to functioning abnormally, will explain the discrepancy between the actual and the intended system behaviour. Note that the resulting diagnoses need not to be unique, i.e. there can be different explanations for a faulty behaviour.

Based on the description of a $\mathcal{PFSA}$=(Q, Σ, Π, E, S, F), a $\mathcal{PFSA}$ Diagnosis Problem can be defined as follows.

Definition 8 ($\mathcal{PFSA}$ Diagnosis Problem): A $\mathcal{PFSA}$ Diagnosis Problem is represented by a tuple (SD, $TRANS$), where $SD = STAT \cup WF$. $STAT$ is the structural description of a $\mathcal{PFSA}$ represented by a set of finite domain variables. WF represents the intended behaviour of a $\mathcal{PFSA}$ which is represented by a set of constraints on $STAT$. Finally, $TRANS$ is a set of transition conditions of the $\mathcal{PFSA}$ represented by a set of constraints on $STAT$. □

The construction of $STAT$ is based on the definition of a set of finite domain variables related to paths of the reachability tree (see Figure 2), e.g. {kl_1, id_1, av_1, sl_1} represent the path p_1. Note that the projection of all solutions for $SD \cup$

	path p_1	path p_2	path p_3	path p_4	path p_5	path p_6	path p_7
var(q_0)=kl	kl_1	kl_2	kl_3	kl_4	kl_5	kl_6	kl_7
var(q_1)=aw				aw_4	aw_5	aw_6	aw_7
var(q_2)=id	id_1	id_2	id_3	id_4	id_5	id_6	
var(q_3)=ds							ds_7
var(q_4)=av	av_1	av_2		av_4	av_5		
var(q_5)=sh		sh_2		sh_4			
var(q_6)=sl	sl_1		sl_3		sl_5	sl_6	

Table 1: Variables representing input sequences of a $\mathcal{PFSA}$, e.g. instantiations of the variables $\{kl_1, id_1, av_1, sl_1\}$ represent possible input sequences for path p_1.

$TRANS$ to {kl_1, id_1, av_1, sl_1} exactly represents those input sequences accepted by path p_1. For our example $\mathcal{PFSA}$, $STAT$ can be defined as follows (see also Table 1).

Example 2 ($STAT$):
$STAT$ = {kl_1,id_1,av_1,sl_1,kl_2,id_2,av_2,sh_2, kl_3,id_3,sl_3, kl_4, aw_4, id_4, av_4, sh_4, kl_5, aw_5, id_5, av_5, sl_5, kl_6, aw_6, id_6, sl_6, kl_7, aw_7, ds_7} □

Note that we introduce an additional value $noval$ to the domain of each variable $\in STAT$. In the case that a path of the reachability tree represents an illegal trajectory, i.e. no consistent value assignment exists for the corresponding variables, all variables of this path have the assignment $noval$ - this is assured by additional meta-constraints which are defined for each path of the reachability tree, e.g. for path p_1: $kl_1 \neq noval \wedge id_1 \neq noval \wedge av_1 \neq noval \wedge sl_1 \neq noval \vee kl_1 = noval \wedge id_1 = noval \wedge av_1 = noval \wedge sl_1 = noval$.

The construction of well-formedness rules related to the *extensibility* of paths within a $\mathcal{PFSA}$ is shown in the following. For each *consistent* path in the reachability tree of a $\mathcal{PFSA}$ we introduce a well-formedness rule as follows. Since e.g. [(q_0,{c_2},q_1)] is a consistent path, a corresponding well-formedness rule is defined as follows.

Example 3 (well-formedness rules for extensibility):
$WF_{extensibility}$ ([(q_0,{c_2},q_1)]) = {
$kl_4 \neq noval \wedge aw_4 \neq noval \vee$
$kl_5 \neq noval \wedge aw_5 \neq noval \vee$
$kl_6 \neq noval \wedge aw_6 \neq noval \vee$
$kl_7 \neq noval \wedge aw_7 \neq noval$}

$WF_{extensibility}$ ([(q_0,{c_2},q_1), (q_1,{c_4},q_3)]) = {
$kl_7 \neq noval \wedge aw_7 \neq noval \wedge ds_7 \neq noval$}
$WF_{extensibility}$ ([(q_0,{c_2},q_1), (q_1,{c_3},q_2)]) = {
$kl_4 \neq noval \wedge aw_4 \neq noval \wedge id_4 \neq noval \vee$
$kl_5 \neq noval \wedge aw_5 \neq noval \wedge id_5 \neq noval \vee$
$kl_6 \neq noval \wedge aw_6 \neq noval \wedge id_6 \neq noval$ }
$WF_{extensibility}$ ([(q_0,{c_2}, q_1), (q_1,{c_3},q_2), (q_2,{c_5},q_6)]) = {
$kl_6 \neq noval \wedge aw_6 \neq noval \wedge id_6 \neq noval \wedge sl_6 \neq noval$}
$WF_{extensibility}$ ([(q_0,{c_2}, q_1), (q_1,{c_3}, q_2), (q_2,{c_6, c_7},q_4)]) = {
$kl_4 \neq noval \wedge aw_4 \neq noval \wedge id_4 \neq noval \wedge av_4 \neq noval \vee$
$kl_5 \neq noval \wedge aw_5 \neq noval \wedge id_5 \neq noval \wedge av_5 \neq noval$}
$WF_{extensibility}$ ([(q_0,{c_1},q_2)]) = {
$kl_1 \neq noval \wedge id_1 \neq noval \vee$
$kl_2 \neq noval \wedge id_2 \neq noval \vee$
$kl_3 \neq noval \wedge id_3 \neq noval$}
$WF_{extensibility}$ ([(q_0,{c_1},q_2), (q_2,{c_5},q_6)]) = {
$kl_3 \neq noval \wedge id_3 \neq noval \wedge sl_3 \neq noval$} □

For each transition (q_i, C_i, q_j) $\in E$ of a $\mathcal{PFSA}$ we can derive a rule guaranteeing that the transition is *accessible* by at least one path. In our example the connection between the states q_0 and q_2 is represented by the variables kl_1, kl_2, kl_3, consequently at least one of those variables must be instantiated with a value $\neq noval$.

Example 4 (well-formedness rules for accessibility):
$WF_{accessability}$ (q_0, {c_1}, q_2) = {
$kl_1 \neq noval \vee kl_2 \neq noval \vee kl_3 \neq noval$.}
$WF_{accessability}$ (q_0, {c_2}, q_1) = {
$kl_4 \neq noval \vee kl_5 \neq noval \vee kl_6 \neq noval \vee kl_7 \neq noval$.}
$WF_{accessability}$ (q_1, {c_4}, q_3) = {
$aw_7 \neq noval$.}
$WF_{accessability}$ (q_1, {c_3}, q_2) = {
$aw_4 \neq noval \vee aw_5 \neq noval \vee aw_6 \neq noval$.}
$WF_{accessability}$ (q_2, {c_6, c_7}, q_4) = {
$id_1 \neq noval \vee id_2 \neq noval \vee id_4 \neq noval \vee id_5 \neq noval$.}
$WF_{accessability}$ (q_2, {c_5}, q_6) = {
$id_3 \neq noval \vee id_6 \neq noval$.}
$WF_{accessability}$ (q_4, {c_8, c_9}, q_6) = {
$av_1 \neq noval \vee av_5 \neq noval$.}
$WF_{accessability}$ (q_4, {c_{10}, c_{11}}, q_5) = {
$av_2 \neq noval \vee av_4 \neq noval$.} □

For each state $q_i \in Q$ - F of the reachability tree, we derive a rule guaranteeing *determinism* for all paths including q_i, e.g. input values of kl related to the paths p_1, p_2, p_3 must be different from those related to the paths p_4, p_5, p_6, p_7 - this property is expressed in a well-formedness rule related to q_0.

Example 5 (well-formedness rules for determinism):[2]
$WF_{determinism}$ (q_0) = {
$(kl_1 \neq kl_4) \wedge kl_1 \neq kl_5 \wedge kl_1 \neq kl_6 \wedge kl_1 \neq kl_7$.
$kl_2 \neq kl_4 \wedge kl_2 \neq kl_5 \wedge kl_2 \neq kl_6 \wedge kl_2 \neq kl_7$.
$kl_3 \neq kl_4 \wedge kl_3 \neq kl_5 \wedge kl_3 \neq kl_6 \wedge kl_3 \neq kl_7$.}
$WF_{determinism}$ (q_2) = {
$kl_1 \neq kl_3 \vee id_1 \neq id_3$.
$kl_2 \neq kl_3 \vee id_2 \neq id_3$.
$kl_6 \neq kl_4 \vee aw_6 \neq aw_4 \vee id_6 \neq id_4$.

[2] In this context we use an abbreviated notation, i.e. instead of ($kl_1 \neq kl_4 \vee kl_1 \neq noval \vee kl_4 \neq noval$) we write $kl_1 \neq kl_4$.

Project	#q_i	#transitions	#diagnoses	#conflicts	computing time (msec.)	computing time QX
1	*6*	*5*	*2*	*2*	*61*	*51*
2	*7*	*8*	*3*	*4*	*230*	*220*
3	*15*	*18*	*2*	*4*	*170*	*130*
4	*27*	*38*	*3*	*6*	*1953*	*1893*
5	*25*	*29*	*2*	*4*	*761*	*701*
6	*63*	*82*	*5*	*15*	*7060*	*7010*

Table 2: Empirical evaluation of diagnosis algorithm. #q_i represents the number of states of the $\mathcal{PFSA}$, *#transitions* represents the number of transitions of the $\mathcal{PFSA}$, *#diagnosis* the number possible diagnosis calculated for the $\mathcal{PFSA}$, *#conflicts* the number of calculated conflicts (QX).

$kl_6 \neq kl_5 \vee aw_6 \neq aw_5 \vee id_6 \neq id_5$. }
$WF_{determinism}$ (q_1) = {
$kl_7 \neq kl_4 \wedge kl_7 \neq kl_5 \wedge kl_7 \neq kl_6 \vee$
$aw_7 \neq aw_4 \wedge aw_7 \neq aw_5 \wedge aw_7 \neq aw_6$.}
$WF_{determinism}$ (q_4) = {
$kl_1 \neq kl_2 \vee id_1 \neq id_2 \vee av_1 \neq av_2$.
$kl_4 \neq kl_5 \vee aw_4 \neq aw_5 \vee id_4 \neq id_5 \vee av_4 \neq av_5$.} □

An example for a corresponding set of transition conditions ($TRANS$) is the following.[3]

Example 6 ($TRANS$ **for** $\mathcal{PFSA}$**)**:

The transition conditions of the $\mathcal{PFSA}$ in Figure 1 are represented as follows.

$TRANS$={
c_1: (kl_1 = $beginner \vee kl_1$ = $noval$) $\wedge$ (kl_2 = $beginner \vee$ kl_2 = $noval$) $\wedge$ (kl_3 = $beginner \vee kl_3$ = $noval$).
c_2: (kl_4 = $expert \vee kl_4$ = $noval$) $\wedge$ ... $\wedge$ (kl_7 = $expert \vee kl_7$ = $noval$).
c_3: (aw_4 = $yes \vee aw_4$ = $noval$) $\wedge$ ... $\wedge$ (aw_6 = $yes \vee aw_6$ = $noval$).
c_4: (aw_7 = $no \vee aw_7$ = $noval$).
c_5: (id_3 = $shortterm \vee id_3$ = $noval$) $\wedge$ ... $\wedge$ (id_6 = $shortterm \vee id_6$ = $noval$).
c_6: ($id_1 \neq shortterm \vee id_1$ = $noval$) $\wedge$ ... $\wedge$ ($id_5 \neq shortterm \vee id_5$ = $noval$).
c_7: ($kl_1 \neq beginner \vee kl_1$ = $noval$) $\wedge$ ... $\wedge$ ($kl_5 \neq beginner \vee kl_5$ = $noval$) .
c_8: (av_1 = $no \vee av_1$ = $noval$) $\wedge$ (av_5 = $no \vee av_5$ = $noval$).
c_9: (kl_1 = $beginner \vee kl_1$ = $noval$) $\wedge$ (kl_5 = $beginner \vee kl_5$ = $noval$).
c_{10}: (av_2 = $yes \vee av_2$ = $noval$) $\wedge$ (av_4 = $yes \vee av_4$ = $noval$).
c_{11}: (kl_2 = $average \vee kl_2$ = $noval$) $\wedge$ (kl_4 = $average \vee kl_4$ = $noval$).}. □

Given a specification of (SD, $TRANS$), a $\mathcal{PFSA}$ Diagnosis can be defined as follows.

Definition 10 ($\mathcal{PFSA}$ **Diagnosis)**: A $\mathcal{PFSA}$ Diagnosis for a $\mathcal{PFSA}$ Diagnosis Problem (SD, $TRANS$) is a set $S \subseteq TRANS$ s.t. $SD \cup TRANS - S$ is consistent. □

A diagnosis exists under the reasonable assumption that $STATS \cup WF$ is consistent.

[3]Note that in the case that transition conditions are not deterministic, we modify (extend) the corresponding transitions which in the following induce conflicts with WF.

Remark: Given a $\mathcal{PFSA}$ Diagnosis Problem (SD, $TRANS$), a Diagnosis S for (SD = $STATS \cup WF$, $TRANS$) exists *iff* $STATS \cup WF$ is consistent. Assuming that SD = $STATS \cup WF$ is inconsistent, it follows from the definition of a diagnosis S that $SD \cup TRANS - S$ is inconsistent $\forall S \subseteq TRANS$. Assuming that $SD \cup TRANS - S$ is consistent, it follows that $STATS \cup WF$ is consistent.

The calculation of diagnoses for a given $\mathcal{PFSA}$ definition is based on the concept of conflict sets.

Definition 11 (Conflict Set): a Conflict Set (CS) for (SD, $TRANS$) is a set $\{c_1, c_2, ..., c_n\} \subseteq TRANS$, s.t. $\{c_1, c_2, ..., c_n\} \cup SD$ is inconsistent. CS is minimal *iff* $\neg\exists$ $CS' \subset CS$: conflict set (CS'). □

The algorithm for calculating a set of minimal diagnoses for a process definition is the following (Algorithm 1). The labelling of the search tree (Hitting Set Directed Acyclic Graph - HSDAG) is based on the labelling of the original HSDAG [Reiter, 1987]. A node n is labelled by a corresponding conflict set $CS(n)$. The set of edge labels from the root to node n is referred to as $H(n)$.

Algorithm 1 $\mathcal{PFSA}$-Diagnosis($SD, CONDS$)

(a) Generate a pruned HSDAG T for the collection of conflict sets induced by transitions of $TRANS$ in breadth-first manner (we generate diagnoses in order of their cardinality). With every theorem prover (TP) call at a node n of T the consistency of ($TRANS$ - $H(n) \cup SD$) is checked. If there exists an inconsistency, a conflict set CS is returned, otherwise ok is returned. If ($TRANS$ - $H(n) \cup SD$) is consistent, a corresponding diagnosis $H(n)$ is found.

(b) Return $\{H(n) \mid n$ is a node of T labeled with $ok\}$.

Minimal conflict sets determined by different TP calls are $\{c_2, c_{11}\}$, $\{c_7, c_9\}$, $\{c_2, c_9\}$, and $\{c_1, c_9\}$. One minimal diagnosis S for our example $\mathcal{PFSA}$ is $\{c_2, c_9\}$. From each solution for $SD \cup TRANS - S$ a corresponding set of repair actions REP can be derived by generalizing variable assignments related to transition conditions of S. E.g. in our example $\mathcal{PFSA}$ the (faulty) transition condition c_2 is represented by the variables kl_4, kl_5, kl_6, kl_7. From $kl_4 = average, kl_5 = noval, kl_6 = average, kl_7 = expert$ (solution related to the diagnosis S=$\{c_2, c_7\}$) we can derive $\{kl = average \vee kl = expert\} \subseteq REP$.

4 Evaluation

The presented diagnosis support has been implemented within the scope of the Koba4MS project. The diagnosis component is a Java-based implementation of the diagnosis algorithm presented in [Reiter, 1987; Greiner *et al.*, 1989]. The calculation of conflict sets is based on the *QuickXPlain* (see *QX* in Table 2) algorithm presented in [Junker, 2004]. Table 2 depicts the results of an empirical evaluation of the implemented diagnosis environment. In general the diagnosis component is applicable to interactive settings, where a knowledge engineer is developing and maintaining a recommender knowledge base. The projects 1-6 shown in Table 2 are related to different application domains (e.g. financial services, digital cameras, notebooks, and financial support advisors) - typical faulty conditions modeled by engineers were introduced into the process definitions.

5 Related Work

The increasing size and complexity of knowledge bases motivated the application of model-based diagnosis (MBD) [Reiter, 1987] in knowledge-based systems development [Felfernig *et al.*, 2004]. Similar motivations led to the application of model-based diagnosis approaches in technical domains such as the development of hardware designs [Friedrich *et al.*, 1999], on-board diagnosis for automotive systems [Sachenbacher *et al.*, 2000] and in software development [Mateis *et al.*, 2000]. Practical experiences related to the implementation of the concepts presented in [Felfernig *et al.*, 2004] are discussed in [Fleischanderl, 2002]. A value-based model of program execution is introduced in a.o. [Mateis *et al.*, 2000], which is an extension of the approach of debugging imperative languages [Friedrich *et al.*, 1999]. Using MBD techniques, the location of errors is based on the analysis of a given source code. Additionally, a set of test cases specifying the expected behaviour of the program is required - this set encompasses concrete values for variables, assertions, reachability constraints for paths, valid call sequences etc. An introduction to the concepts of MBD diagnosis and an overview of existing applications can be found in [Peischl and Wotawa, 2003] an overview on the application of model-based diagnosis techniques in software debugging can be found in [Stumptner and Wotawa, 1998].

6 Conclusions

In this paper we have presented debugging concepts implemented in a debugging environment for knowledge-based recommender applications. Predicate-augmented finite state automata ($\mathcal{PFSA}$) are applied to represent models of possible user interactions. A $\mathcal{PFSA}$ is translated into a corresponding representation of a constraint satisfaction problem, which is used in order to calculate diagnoses related to faulty conditions in the process definition.

References

[Bass and Coutaz, 1991] L. Bass and J. Coutaz. *Developing Software for the User Interface*. The SEI Series in Softw. Eng. Addison Wesley, Massachusetts, USA, 1991.

[Felfernig *et al.*, 2004] A. Felfernig, G. Friedrich, D. Jannach, and M. Stumptner. Consistency-based Diagnosis of Configuration Knowledge Bases. *AI Journal*, 2(152):213–234, 2004.

[Fleischanderl, 2002] G. Fleischanderl. Suggestions from the software engineering practice for applying consistency-based diagnosis to configuration knowledge bases. In *13th International Workshop on Principles of Diagnosis (DX-02)*, 2002.

[Friedrich *et al.*, 1999] G. Friedrich, M. Stumptner, and F. Wotawa. Model-based diagnosis of hardware designs. *AI Journal*, 111(2):3–39, 1999.

[Greiner *et al.*, 1989] R. Greiner, B.A. Smith, and R.W. Wilkerson. A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, 1989.

[Hopcroft and Ullman, 1979] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, Massachusetts, USA, 1979.

[Junker, 2004] U. Junker. QUICKXPLAIN: Preferred Explanations and Relaxations for Over-Constrained Problems. In *19th National Conference on AI (AAAI04)*, pages 167–172, San Jose, California, 2004.

[Mateis *et al.*, 2000] Mateis, M. Stumptner, and F. Wotawa. Modeling Java programs for diagnosis. In *14th European Conference on Artificial Intelligence*, pages 171–175, Berlin, Germany, 2000.

[Peischl and Wotawa, 2003] B. Peischl and F. Wotawa. Model-Based Diagnosis or Reasoning from First Principles. *IEEE Intelligent Systems*, 18(3):32–37, 2003.

[Reiter, 1987] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 23(1):57–95, 1987.

[Sachenbacher *et al.*, 2000] M. Sachenbacher, Pr. Struss, and C.M Carlen. A Prototype for Model-Based On-Board Diagnosis of Automotive Systems. *AI Communications*, 13(2):83–97, 2000.

[Stumptner and Wotawa, 1998] M. Stumptner and F. Wotawa. A Survey of Intelligent Debugging. *European Journal on Artificial Intelligence (AICOM)*, 11(1):35–51, 1998.

[v.Noord and Gerdemann, 2001] G. v.Noord and D. Gerdemann. Finite State Transducers with Predicates and Identities. *Grammars*, 4(3):263–286, 2001.

[Wasserman, 1985] A. Wasserman. Extending state transition diagrams for the specification of human-computer interaction. *IEEE Transactions on Software Engineering*, 11:699–713, 1985.

A Conceptual Model for Configurable Services*

Mikko Heiskala[1], Juha Tiihonen[2], and Timo Soininen[2]
[1)] BIT Research Centre & [2)] Software Business and Engineering Institute (SoberIT)
Helsinki University of Technology
[1)]P.O. Box 5500, [2)] P.O. Box 9210, FI-02015 HUT, Finland
firstname.lastname@tkk.fi

Abstract

We present a conceptual model for modeling configurable services, an area with relatively little previous research, based on a synthesis of previous work extended with our own experiences.

1 Introduction

In the past decades several configurator systems have been developed, mostly for mechanical and electronics products. [e.g. Faltings and Freuder, 1998; Soininen and Stumptner, 2003]. The central issues for such systems have been suitable product knowledge modeling concepts and formal languages based on these, and correct and efficient inference algorithms for supporting the configuration tasks [e.g. Faltings and Freuder, 1998; Soininen *et al.*, 1998; Felfernig *et al.*, 2001; Soininen and Stumptner, 2003]. However, there is relatively little research on *configurable* or *mass customizable services,* and on developing configurators particularly suitable for these [Harvey *et al.*, 1997, Da Silveira *et al.*, 2001; Papathanassiou, 2004; Paloheimo *et al.*, p. 41, 62, 2004; Akkermans *et al.*, 2004; Wimmer *et al.*, 2003; Peters and Saidin, 2000; Winter, 2001; Meier *et al.*, 2002; Dausch and Hsu, 2003; Böhmann *et al.*, 2003].

In this paper we provide one step towards tool support by presenting a conceptual model for modeling the knowledge important for configuring services in a configurator, from the points of view of the customer and sales person. Our conceptual model is an extension of a synthesis of the few existing proposals for modeling configurable services [Akkermans *et al.*, 2004; Wimmer *et al.*, 2003], based on our experiences on how such services could be modeled in four case companies that we have been working with for the past one-and-a-half years. The companies represent machinery maintenance, insurance and telecommunications services.

The rest of the paper is structured as follows. Section 2 discusses the central literature and definitions on configurable services and approaches for modeling them. We present the Four-Worlds Model for Configurable Services in Section 3 and discuss and compare it to previous work in Section 4. Finally, we present our conclusions and directions for future work in Section 5.

2 Configurable services

Here we briefly discuss the central literature and definitions on configurable services and approaches for modeling them.

Service has been defined as a process, in which customers often participate, carried out by the provider as a solution to customer problems. Services often are intangible, perishable, simultaneously produced and consumed, and heterogeneous – all characteristics goods rarely have [Grönroos, p. 46-7, 2000]. Due to these differences, simple adoption of modeling approaches from goods may not be appropriate [Wimmer *et al.*, 2003], as how well a conceptualization fits its intended domain of use affects its quality. For the purposes of this paper we define *configurable services* as services that can be customized to individual specifications from a set of options designed to meet a pre-determined range of customer needs.

The need for configurable services stems from the raise in service customers' demand for services that meet their increasingly diverse needs better [Harvey *et al*., 1997; Papathanassiou, 2004; Wimmer *et al.*, 2003]. It has been suggested that service companies should meet this demand by customizing their services [Harvey *et al.*, 1997] by mixing and matching service modules to meet customer needs [Meyer and DeTore, 2001; Peters and Saidin, 2000]. This seems similar to the aims of product configuration. This kind of *service configuration,* possibly supported with configurators has been at least a partial goal in several papers. Of these, Winter [2001], Meier *et al.* [2002], Böhmann *et al.* [2003], and Jiao *et al.* [2003] do not define a conceptualization for services but do discuss service modeling issues. Dausch and Hsu [2003] propose a reference model for mass customizing maintenance services not directly intended for configuration and possibly lacking generalizability with a focus on one domain. Akkermans *et al.* [2004] and Wimmer *et al.* [2003] present conceptual models of configurable services and thus resemble our work most closely, and are hence the only ones discussed in more detail here.

Akkermans *et al.* [2004] present a conceptual model aimed at facilitating configuration of services in a Semantic Web Services environment. The model describes services

* The support of National Technology Agency of Finland, project 40370/03, its participating companies, and Research Foundation of Helsinki University of Technology is gratefully acknowledged.

from three perspectives: *value* (customer)*, offering* and *process perspectives*. Wimmer *et al.* [2003] propose a model for modeling financial services in a mass customization setting. The models of Akkermans *et al.* [2004] and Wimmer *et al.* [2003] seem to capture services a bit differently. Wimmer *et al.*'s [2003] model roughly corresponds to Akkermans *et al.*'s [2004] offering perspective. Wimmer *et al.* [2003] do not model similar issues as Akkermans *et al.*'s [2004] value and process perspectives. Lacking a process viewpoint might be a potential weakness as services are deemed to be processes that customer participation may affect [Grönroos, p. 47, 2000; Akkermans *et al.*, 2004; Böhmann *et al.*, 2003; Jiao *et al.*, 2003]. According to Wimmer *et al.* [2003], Winter [2001], and our findings in our insurance company case, capturing customer characteristics seems to be significant for services. However, neither Wimmer *et al.* [2003] nor Akkermans *et al.* [2004] provide specific concepts for customer characteristics. Further, from our maintenance service cases it seems that the characteristics of the maintained machinery may affect the service agreement as well. Therefore, it seems that neither model captures all aspects potentially relevant for services.

3 Four-Worlds Model for Configurable Services

Here we present the Four-Worlds Model for Configurable Services with a configuration modeling focus. We use the Unified Modeling Language (UML) [e.g. Booch *et al.*, 1999] to define the model semi-formally. The concept of using the four worlds to describe configurable services on a more general level will be described in a later publication.

3.1 Overview of the model

The model is aimed to capture configurable services at sufficient and appropriate detail for the sales stage. The model is divided to four viewpoints we call *worlds*, see Figure 1 (a), each having their own main concepts and dependencies between them. The *objects-of-service world* describes the recipient(s) of service (like persons or physical systems) and the environment relevant to the recipient(s). The objects-of-service world aims to specify *what the company needs to know* about the service recipients and the environment to be able to successfully configure the service. The *needs world* captures the reasons *why* a customer would want to buy the service. The *service solutions world* describes the specifications to which the service is to be delivered, i.e. *what* is to be delivered. The *process world* depicts the service delivery process and resources required in it at appropriate detail for the sales stage. It is intended for communicating the process for the customer (possibly participating in it), not for detailing operation procedures for the company employees. The process world describes *how and with what* the service is put into practice.

There can be dependencies between the worlds. A service recipient with given characteristics has certain needs and requires a specific service solution. The needs are satisfied with particular solutions. The solutions are carried out with given processes and involve certain resources. The service recipient may be a more or less active participant in the process, thus being required as a resource in the process.

Example: We demonstrate the conceptual model using an example loosely based on insurance services of one of our case companies, Tapiola Group[1], a prominent group of Finnish mutual insurance and financial services companies. Tapiola's use of a tool helping insurance clerks to identify relevant customer characteristics and needs and subsequently recommend suitable insurances based on these has in part motivated our objects-of-service and needs worlds.

The objects-of-service world entails a customer and a car. The age of the car must be known as it affects the possible insurance coverage and may affect needs.

The needs world describes motoring-oriented needs of the customer. In case of an accident or a breakdown, it is possible to specify the desired level of assistance and whether collision damages should be covered.

The service solutions world consists of a car insurance solution that includes mandatory car insurance, and optionally one of two types of voluntary car insurance. Both include coverage against theft, fire, etc., and collision damage coverage is optional. Budget voluntary car insurance is available only for cars that are at least six years old. It is less expensive, because cheapest accepted (third-party) parts are used for repairs and the insurance company decides where and when the car will be repaired. Extended voluntary car insurance is available only for cars that are 10 years old at the most. It has optional road assistance with two levels, both levels cover towing costs and the insurance company arranges towing. Extended road assistance adds compensation for expenses to continue the trip.

The process world defines a process in case of a car breakdown or an accident. If towing is needed, the insurance company arranges it if the car has extended voluntary car insurance. Otherwise the customer must arrange the towing himself. After the towing the car must be repaired. The repair process depends on the type of the voluntary car insurance. The budget repair process is applied for cars with budget voluntary car insurance, and the normal repair process is applied to cars with extended voluntary car insurance.

3.2 General modeling concepts

In this section we introduce the general modeling mechanisms. A *configuration model* defines a configurable service and the possibilities of customizing it with *types* and their *properties*. We distinguish between *TYPES* and their instances occurring in a *configuration*, i.e. *INDIVIDUALS*. A configuration describes a specific service instance.

We use UML in the metamodel (Figure 1 (b)) to define stereotypes corresponding to the modeling concepts. Configuration models contain classes that are instances of these stereotypes (Figure 1 (c)). A configuration contains instances of classes of a configuration model (Figure 1 (d)). We left out '« »' denoting a stereotype from UML association names in Figures 1 (c) and 1 (d) due to limited space.

[1] Tapiola Group, http://www.tapiola.fi/wwweng/briefly/

Types can be organized in *generalization* hierarchies. A *subtype* inherits the properties, i.e. *parts*, *attributes*, and *constraints*, of its *supertypes*. Subtypes are said to be *direct subtypes* of the supertypes directly above them in the generalization hierarchy. Direct supertypes are defined analogously. Types describe their compositional structure with parts, specified with *PART DEFINITIONS* that define a *part name*, a *set of possible types* that can occur as the part, and a *cardinality* describing the number of individuals that must occur as the part. Additionally, parts of process module types define their possible *successors* (see subsection 3.6). *ATTRIBUTES* characterize types. A type defines for its attributes an *attribute name* and the *possible values* the attribute can have through its *VALUE TYPE*. Types define *CONSTRAINTS* that specify conditions that must hold in a correct configuration. Constraints can be used to model arbitrarily complex interdependencies of types, individuals and their properties when other mechanisms in our model are not sufficient to capture an aspect of a service. Constraints are either *hard* or *soft*. A hard constraint must always hold whereas a soft con-

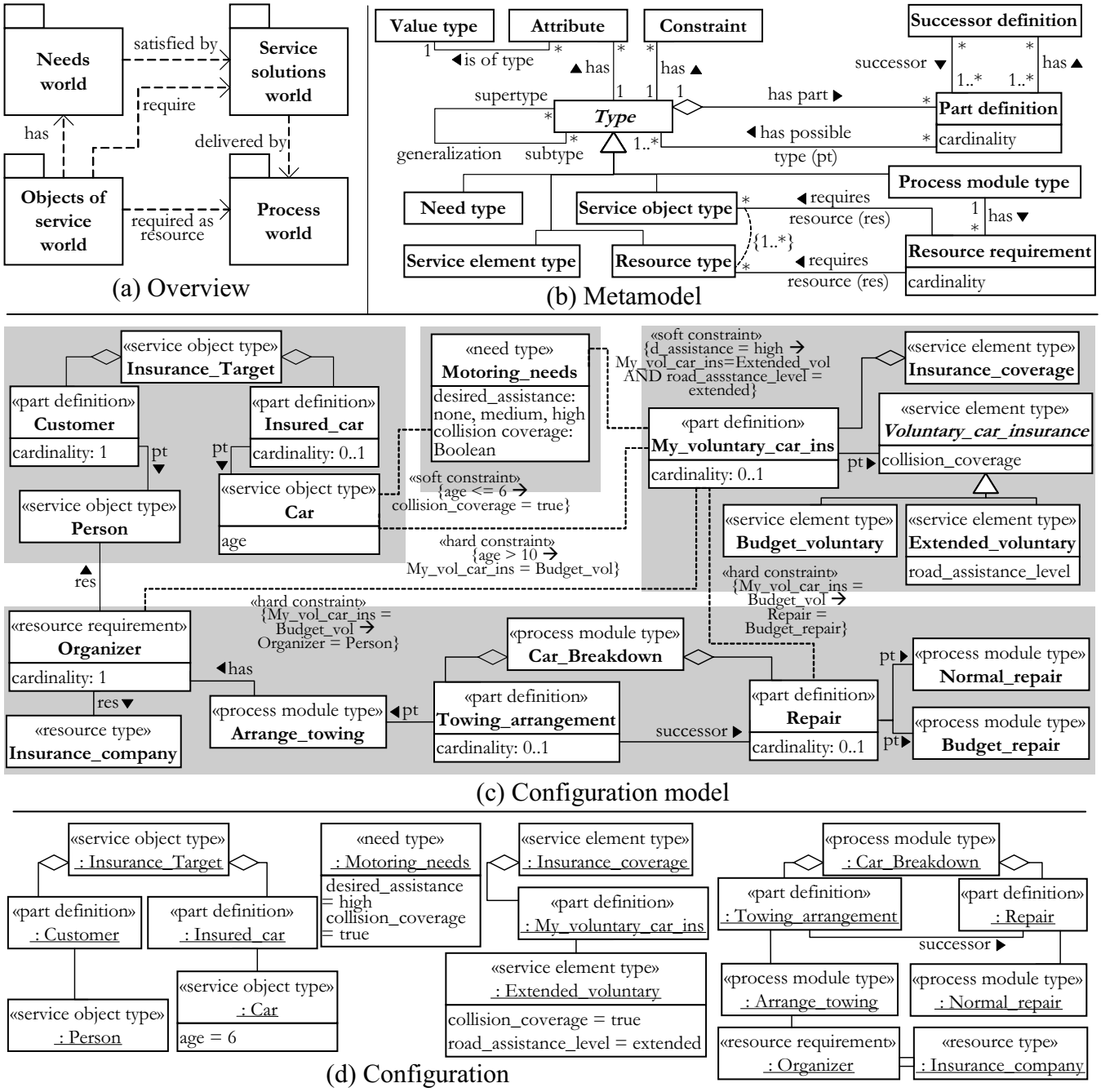


Figure 1 (a) Overview, (b) Metamodel, (c) Example configuration model, (d) Example configuration

straint can be violated. We assume there is a constraint language with sufficient expressive power.

There are some general modeling restrictions in our model. Types in a generalization hierarchy must be of the same direct subtype of *Type*, e.g. all supertypes of a need type must be need types as well. Types in a compositional structure must be of same direct subtype of *Type*, e.g. parts of a need type must have only need types as possible types.

3.3 Objects-of-service world

The main concept of the objects-of-service world is *service object*. A *SERVICE OBJECT TYPE* is an entity representing a service recipient (like persons or physical systems) or environment relevant to the recipient. Examples for the compositional structure of service objects could be a family and its members or maintained equipment and its structure. ***Example:*** Figure 1(c) shows the configuration model of our running example, and Figure 1 (d) exhibits a configuration. *Insurance target* represents objects-of-service. It contains exactly one *Person* in the role of a *Customer*. The role of an *Insured_car* specifies properties of the customer's car to be insured. The configuration has a *Person,* and a *Car* with 6 years of *age*.

3.4 Needs world

Need is the main concept in the needs world. A *NEED TYPE* denotes a benefit sought from a service by a customer. Basis for compositional structure of needs can be e.g. decomposing general needs to more detailed ones, like a general need of being reachable at all times decomposed to being reachable by phone, fax, or email.

Example: The needs world describes if collision damages coverage is desired, and the level of assistance in case of an accident or a breakdown. This is modeled with attributes of *Motoring_needs*. The example *Motoring_needs* specify *high desired_assistance,* and *collision coverage* is included.

3.5 Service solutions world

The service solutions world is centered on a *service element*. A *SERVICE ELEMENT TYPE* describes a part of the pre-delivery service specification, i.e. agreement, about what is to be delivered. Examples of service elements and their compositional structure could be messaging services decomposed to SMS, MMS, fax, and email messaging.

Example: Insurance coverage has optionally a *Voluntary_car_insurance* that can be of type *Budget_voluntary* or *Extended_voluntary*. *Extended_voluntary* can have road assistance in case of a car breakdown or an accident. Road assistance is available only for cars whose age is 10 years at the most. *Budget_voluntary* does not include road assistance, and it is available only for cars that are 6 years or older. Accident repairs are made with non-original parts in a repairs shop decided by the insurance company. In *Extended_voluntary* the customer decides where the car is repaired and original parts are be used. Mandatatory car insurance is not modeled due to limitations of space. The configuration contains *Insurance_coverage* with *Extended_voluntary* with included *collision_coverage* and *extended road_assistance_level*.

3.6 Process world

The central concept of the process world is *process module*. A *PROCESS MODULE TYPE* represents a task, which could be carried out as part of the service delivery process. A process module may require specific *resources* to be successfully carried out. In its *RESOURCE DEFINITION*, a process module defines a *set of possible types* that can appear as a resource and a *cardinality* describing the number of individuals that must appear as resources. These resources may be service object types (from the objects-of-service world) or *RESOURCE TYPES.* A resource type describes a physical thing, information, a person, or something else that is necessary for the execution of process modules.

In the process world part definitions take on added semantics: the *precedence* of tasks in a process is defined with part definitions and their *successors*. A part definition defines in its *SUCCESSOR DEFINITION* the part definitions that can follow it in the process. All successors defined in a successor definition must be part definitions of the same process module type. A successor definition can determine the conditions according to which the successor(s) should be carried out, e.g. only one successor or multiple in parallel. The semantics of the compositional structure of process modules is that the execution of a process module individual means that the process module individuals as its parts are executed as well. For example, machinery repair could decompose to parts Notify of fault, Identify fault, Obtain spare parts (either from customer managed on-site stock, if available, or maintenance engineer's own supply), and Repair fault. Of these, Notify of fault could require the customer as a resource depending on whether the maintained machinery has remote fault diagnostics installed or not.

Example: Either the insured or the insurance company arranges towing. Thus, *Arrange_towing* requires as *Organizer* resource either *Insurance_company* or a *Person* denoting the insured. Arrangement of towing can be succeeded in the process with *Repair*, which is either *Budget_repair* or *Normal_repair* depending on the chosen voluntary insurance type. The configuration has *Towing_arrangement* with *Insurace_company* as the Organizer resource, and *Repair* is managed with *Normal_repair* process. We have modeled successor definitions in Figures 1 (c) and 1 (d) only with a named association due to limitations of space.

3.7 Dependencies between worlds

The dependencies are modeled with constraints. Examples can be found in Figure 1 (c).

Objects-of-service – needs world: Certain service objects, or service objects with given properties, *have* specific needs. For example, a single person has different insurance needs compared to a parent with a family to care for. ***Example:*** *collision_coverage* is recommended for cars that are at most six years old.

Objects-of-service – service solutions world: Further, similarly as above, service objects with given properties *require* certain solutions, i.e. service elements. For example,

a customer owning a boat often requires a boat insurance whereas a customer without a boat does not. ***Example:*** The *age* of the *Insured_car* affects available service elements, e.g. *Extended_voluntary* insurance is available only for cars whose *age* is at most 10 years.

Needs – service solutions world: Needs are *satisfied by* specific service elements. For example, a need of being reachable at all times is satisfied by SMS, MMS, and email access with mobile phone services. ***Example:*** If customer desired high level of assistance *Extended_voluntary* is recommended, this is modeled with a soft constraint.

Service solutions – process world: Specific service elements with given properties are *delivered by* certain process modules. For example, a mobile voice mail service is delivered by the company enabling the service at their end and then by customer taking it in use, e.g. initializing passwords. ***Example:*** If *My_voluntary_car_ins* is of type *Budget_voluntary*, *Repair* uses the *Budget_repair* process.

Objects-of-service – process world: Service objects are *required as resources* for process modules. For example, as above, a customer is required to perform actions to take a mobile voice mail in use, like initializing passwords. ***Example:*** A *Person* denoting the customer is required as *Organizer* resource, if *My_voluntary_car_ins* is of type *Budget_voluntary.*

4 Discussion and comparison with previous work

The Four-Worlds Model for Configurable Services still is at an early stage of development. We have not yet carried out in-depth full-scale case studies of modeling and configuring services. Hence we present the model as an idea to be further developed and rigorously tested. Due to this research being at an early stage, the discussion here is geared towards previous work. Nevertheless, in our view the model provides a preliminary synthesis of previous models, extended and simplified according to our experiences. Based on the initial feedback from our cases it would seem that the proposed conceptualization allows modeling these services in a uniform way. Further, the model appears to support structured thinking about configurable services in the companies, which could prove helpful for communication, documentation, and design as well.

We hold the view that the division of the model to the four worlds is beneficial in several ways. In addition to the separation of concerns when modeling, the worlds allow for some flexibility. If modeling a world is not necessary for the service in question it can be left out. However, the solutions world will probably always be present. During configuration, the worlds could be used to phase and direct the process. From another viewpoint, a model with analogous worlds could be used for mechanical products as well. Customer characteristics and needs may influence preferred product options in mechanical products and could perhaps help in eliciting what customers truly require from the product. Capturing the manufacturing process might be relevant for products whose configuration decisions depend on manufacturing constraints like availability of components.

To the best of our knowledge, no previous model of configurable services provides specific concepts to model customer characteristics, an issue considered important in [Wimmer *et al.*, 2003; Winter, 2001]. Further, from our cases it seems that service agreements can be affected by e.g. characteristics of maintained equipment or environment of the service recipient as well. Therefore we see objects-of-service world as an extension to previous approaches.

Previous work contains similar ideas to our needs world. In [Akkermans *et al.,* 2004] the value perspective includes a hierarchy of *customer demands*. Both Soininen *et al.* [1998] and Felfernig *et al*. [2001] use *functions* to denote what benefits or functionality a product offers to its customers distinguishing from concrete product parts. An approach of defining *customer requirements or benefits* to be met with service processes is present in [Jiao *et al.*, 2003; Dausch and Hsu, 2003; Meier *et al.*, 2002]. However, in our view our model provides more variability modeling mechanisms for needs than other service-oriented approaches, like constraints to denote dependencies between needs.

Other approaches appear to capture similar issues as our service solutions world. This seems reasonable as it describes the service agreement, which is what is configured at the sales stage judging from our service cases. There are differences also. Wimmer *et al.* [2003] define *roles* for grouping related *attributes* and *views* to provide different viewpoints to the product information in the model, e.g. configuration or after sales support. In their offering perspective Akkermans *et al.* [2004] use *input* and *outcome ports* of different types of *resources* to model connections between their *service elements* that can be used to model process-like precedence of elements. In [Jiao *et al.*, 2003; Dausch and Hsu, 2003; Meier *et al.*, 2002] the components of agreements seem to represent processes or their outputs. We, however, aim to clearly distinguish between the agreement options and the processes necessary to deliver them. In our service cases agreement options generally do not directly represent processes or their outputs.

Means to capture processes with precedence are mentioned in [Akkermans *et al.*, 2004] albeit not discussed in detail. Dausch and Hsu [2003] describe several processes specific to maintenance services. Modeling the process is seen as important in [Jiao *et al.*, 2003; Meier *et al.*, 2002] and the significance of customer participation in it in [Akkermans *et al.*, 2004; Jiao *et al.*, 2003; Böhmann *et al.*, 2003]. Wimmer *et al.* [2003] do not have process modeling mechanisms perhaps due to their focus on financial services. Our aim in modeling processes is to be able to communicate the process for the potential customer at sales stage. This could help to better manage customers' participation in the process and keep customers' expectations of the service realistic, both possible contributors to better service quality [Grönroos, p. 111-2, 2000]. More complex modeling than ours may be needed if detailed information on the scheduling and production of the service are required.

We use compositional structure and generalization as do Soininen *et al.* [1998] and Felfernig *et al.* [2001]. In fact,

only our process world differs from their models in terms of these modeling mechanisms. However, we are yet to come across a need for resource production and consumption and connections they use for physical products. Thus, our resource concept is not the same as in [Soininen *et al.*, 1998; Felfernig *et al.*, 2001]. Akkermans *et al.* [2004] do not have generalization, differing from us and [Wimmer *et al.*, 2003; Soininen *et al.*, 1998; Felfernig *et al.*, 2001].

5 Conclusions and future work

We have proposed a conceptualization for modeling the knowledge important for configuring services in a configurator, from the points of view of the customer and sales person, taking a step towards filling a gap in configuration research. The conceptualization provides a preliminary synthesis of previous models, extended and simplified according to our experiences. However, we are yet to carry out in-depth full-scale case studies of modeling and configuring services or to implement tool support. Nevertheless, on the basis of initial feedback from our case companies the conceptualization seems suitable for configuration modeling of their services. We have kept what constitutes a correct and complete configuration and pricing out of the scope of this paper, both issues in need of future work.

Acknowledgments

The concept of using the four worlds to describe configurable services this model is based on has been developed by K-S. Paloheimo, J-M. Nurmilaakso, A. Anderson, and the authors. We are grateful for the anonymous reviewers for comments that helped us to improve this paper and for Tapiola for allowing us to use their services in the example.

References

[Akkermans *et al.*, 2004] Akkermans, H., Baida, Z., Gordijn, J., Peña, N., Altuna, A., Laresgoiti, I., "Value Webs: Using Ontologies to Bundle Real-World Services", *IEEE Intelligent Systems*, 19(4):57-66, 2004.

[Booch *et al.*, 1999] Booch, G., Rumbaugh, J., Jacobson, I., The Unified Modeling Language User Guide, Addison-Wesley, 1999.

[Böhmann *et al.,* 2003] Böhmann, T., Junginger, M., Krcmar, H., "Modular service architectures: a concept and method for engineering IT services", *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03)*, pp. 74-83, 2003.

[Da Silveira *et al.*, 2001] Da Silveira, G., Borenstein, D., Fogliatto, F.S., "Mass customization: Literature review and research directions", *Int. J. of Production Economics,* 72(1):1-13, 2001.

[Dausch and Hsu, 2003] Dausch, M. and Hsu, C., "Mass-Customize Service Agreements for Heavy Industrial Equipment", *IEEE International Conference on Systems, Man and Cybernetics,* 5:4809-4814, 2003.

[Faltings and Freuder, 1998] Faltings, B. and Freuder, E.C., "Configuration (Guest editor introduction)", *IEEE Intelligent Systems & Their Applications,* 13(4):32-3, 1998.

[Felfernig *et al.*, 2001] Felfernig, A., Friedrich, G.E., Jannach, D., "Conceptual modeling for configuration of mass-customizable products", *Artificial Intelligence in Engineering,* 15(2):165-176, 2001.

[Grönroos, 2000] Grönroos, C., Service Management and Marketing: A Customer Relationship Management Approach, 2nd Edition, John Wiley & Sons, Ltd., 2000

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

[Jiao *et al.*, 2003] Jiao, J., Ma, Q., Tseng, M.M. "Towards high value-added products and services:mass customization and beyond", *Technovation*, 23(10):809-821, 2003.

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

[Meyer and DeTore, 2001] Meyer, M.H. and DeTore, A., "PERSPECTIVE: Creating a platform-based approach for developing new services", *The J. of Product Innovation Management,* 18(3):188-204, 2001.

[Paloheimo *et al.*, 2004] Paloheimo, K-S., Miettinen, I., Brax, S., Customer Oriented Industrial Services, Helsinki University of Technology, BIT Research Centre, ISBN 951-22-6867-1, Redfina, 2004.

[Papathanassiou, 2004] Papathanassiou, E.A., "Mass customisation: management approaches and internet opportunities in the financial sector in the UK", *Int. J. of Information Management*, 24(5):387-399, 2004.

[Peters and Saidin, 2000] Peters, L. and Saidin, H., "IT and the mass customization of services: the challenge of implementation", *Int. J. of Information Management,* 20(2):103-119, 2000.

[Soininen *et al.*, 1998] Soininen, T., Tiihonen, J., Männistö, T., Sulonen, R., "Towards a General Ontology of Configuration", *AI EDAM*, 12(4):357-372, 1998.

[Soininen and Stumptner, 2003] Soininen, T. and Stumptner, M., "Guest Editorial", *AI EDAM*, (Special Issue on Configuration), 17(1):1-2, 2003.

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

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

Modeling Multiple System Families

Esther Gelle
ABB Switzerland, Corporate Research
5405 Baden-Dättwil
esther.gelle@ch.abb.com

Abstract

We discuss in this paper product models for technical systems that are configured. The challenge is to carefully define a model that is as stable as possible over several system families. For this reason we propose a functional model of the technical system and we discuss the advantages and disadvantages in use and maintenance of two types of generic product models across multiple system families: the superset and the intersection product model. In the end of this paper we briefly present modeling issues related to multi-functional components.

1 Introduction

While configurators are well-established in mass-customization markets, assemble-to order (ATO) and engineer-to-order (ETO) companies have only recently started to look into the use of configurators for their products and systems. Even for individual very customer-specific technical system configurations manufacturers increasingly feel the need to streamline their offer process. With increasing market demand it becomes important to generate offer documentation in a reliable, consistent and fast manner. ATO and ETO technical systems differ from typical configurable products such as cars or computers in the following way:

- Their configurations are less standardized, i.e. a customer specification often results in a unique system configuration.
- Two configurations of a technical system distinguish themselves not only in the way they are assembled but also in their geometrical layout.
- Although each configuration varies from the other, a set of typical configurations can be found (some commonalities or patterns are apparent). In other words, reusable modules can be identified. The repeated alignment of these modules may lead to an unbounded or open configuration space. As an example piers are repeated across a bridge.
- Not only discrete choices are made but also choices in real-valued domains. Geometrical constraints impose conditions on placement in a 2D or 3D space. This results in an unbounded configuration space typical for geometrical constraints.
- The manufacturing of technical system is such that components or sets of components are standardized for manufacturing purposes. The notion of product lines is adopted. A product line describes a family of products or systems which are composed of standard reusable components and have a predefined structure. When speaking about technical systems we prefer the term system family over product line in order to emphasize the characteristics of technical systems.
- Individual configurations interact with their environment, in other words constraints are imposed upon the system through the environment in which it is placed.

We would like to discuss in this paper the way such systems could be modeled for use in a configurator. There exists abundant literature on the subject of product models for single system families. However, in we encountered the needs for configurators covering multiple system families. We are interested in the question if a generic product model for several system families makes sense from the point of view of implementation, use and maintenance of a configurator for a technical system and how such a generic product model can be defined.

2 Configuring Technical Systems

A technical system is defined as an artifact which consists of a number of parts that interact with each other. According to [Simon, 1981] an artifact is an entity that is synthesized by man to obey some goal or purpose. The artifact can be thought of as an interface between an inner environment, the substance and the organization of the artifact itself, and an outer environment, the surroundings in which it operates [Simon, 1981]. The description of an artifact in terms of its organization and its function - its interface between inner and outer environment - is an activity well-known to engineers and designers.

In order to illustrate previous definitions, we introduce an example from the domain of civil engineering, namely bridge design. A bridge designer receives a specification of an outer environment in which a bridge should be built in form of a geometrical and a qualitative description. The geometrical description consists of a two-dimensional section draw-

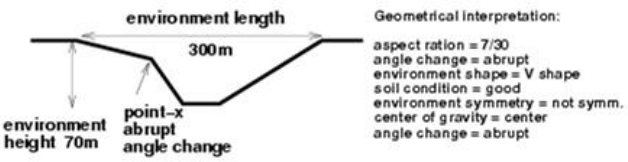


Figure 1: *The environment of a bridge is given by a section view.*

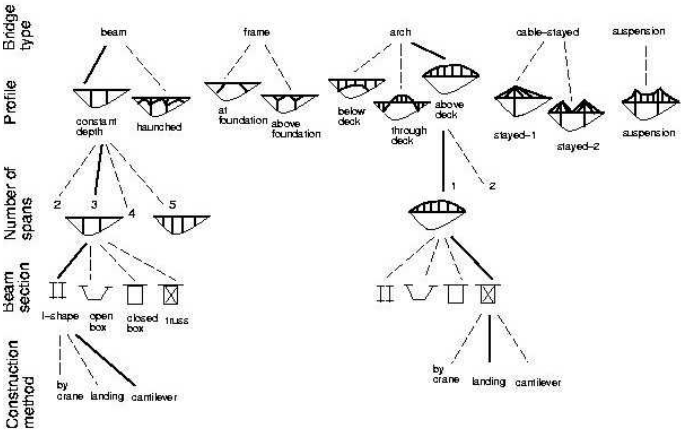


Figure 2: *Different types of bridges as possible solutions.*

ing typically along x-z coordinates. This layout describes for example a valley over which a bridge should be built (Figure 1). In addition a qualitative description includes the characteristics of soil and zones where it is not possible to place a support, for example because a river flows through. The environment also possesses a general characterization in terms of the less immediate surroundings, for example if the valley is situated in the countryside, nearby a city or if the region is known for earthquakes. Further, the designer needs to know the purpose of the bridge. A bridge for a train will obey different dimensioning rules than a bridge intended for pedestrian traffic. On a more general level the designer needs to satisfy country-specific safety and construction rules, he might need to obey governmental regulations regarding esthetic aspects and he may be specialist in the construction of specific types of bridges. The latter simply refers to the accumulated knowledge in bridge design. This experience might be reflected in preferred solutions as shown in Figure 2.

In the context of bridge design, we think of the bridge as a technical system rather than a finished product. In Figure 2 five different bridge types are shown, which correspond to five different system families.

The entire purpose of conceptually modeling a technical system is to support and automate as much as possible the offer process. The input to the technical system has been already described in the bridge example. It consists of a description of the environment - often containing geometrical as well as qualitative elements and a description of the purpose of the artifact itself, e.g. to support a load of vehicles or pedestrians in the example of the bridge and to cross an obstacle (a valley, another road etc.). The description of the outer environment is completed with normative elements of the description such as environment regulations, construction codes etc. The result of the offer process is a set of components that can be manufactured and built together in a way to satify the specification of the outer environment and the purpose or goal of the system. Eventually, the result is documented by:

- A bill of material (BOM) defined as a set of components that can be manufactured.
- The total price including a price list for each component or for sets of components.
- An offer letter.
- Technical data sheets which contain a description of the technical characteristics for each component and for the entire system.
- CAD drawings for the chosen configuration including three dimensional and section views.

The fundamental assumption for automating the offer process for a technical system using a configurator is that the technical system itself is designed in a modular way and can be manufactured based on modular principles. In this sense we assume the presence of system families as a result of the design process for a given product portfolio. A system family can be defined as a family of technical systems that share principles on inner workings - the inner environment - and that also correspond to specific customer requirements or functions. The functional aspects of a technical system are very important, those are the requirements specified by the customer which can be captured independently of a specific inner structure and which represent desired attributes of the system itself. Often customer requirements do not specify technical details rather expected functions of the system and systemic aspects such as performance, stability or reliability. A functional representation of a technical system is thus expected to be more stable than the system families embodying those functions.

There is abundant literature on the subject of product modeling for software as well as for hardware products. Formal theories [Salustri, 1996] and graph theory [Shai and Preiss, 1999] have been used for representing engineering systems in manufacturing and design. In product configuration, [Göpfert and Steinbrecher, 2000], [Najman and Stein, 1992], [Pernler and Leitgeb, 1996], [Soininen *et al.*, 1998], and [Mittal and Frayman, 1989] emphasize the importance of a functional representation. [Göpfert and Steinbrecher, 2000] describe a method for system building in a product development environment. This methods proceeds from the description of customer requirements using a functional decomposition to a component and finally to a manufacturable module view. In essence, two hierarchies are represented: a functional and a component-oriented hierarchy. Elementary functions relate to components through a possibly n-to-n mapping. In this context they propose to build the organization around the product development process to improve efficiency. [Najman and Stein, 1992] use a functionality-centered approach

where functions are composed through selecting object properties in order to specify the functionality of the entire system. Using a formal representation they show that the resource- and the structure-oriented model of configuration problems are equivalent in terms of expressivity of knowledge. [Pernler and Leitgeb, 1996] describe a language that allows for modeling reasoning as well mapping tasks of functional objects. The mapping part deals with mapping functional attributes and objects to structural objects while functional decomposition and value assignments are part of the reasoning task. [Soininen *et al.*, 1998] provide a general ontology for describing configuration models and solutions. They view functions as distinct concepts to convey meaning to the final user of a product in the process of sales configuration. The proposed ontology models function types, individual functions, a part-of function hierarchy, function attributes as well as specification constraints which restrict the combinations of functions that a product can implement. Implementation constraints then describe the mapping from functions to technical concepts. [Mittal and Frayman, 1989] impose some restrictions on the general configuration task in form of functional architecture and the concept of key components. A functional architecture describes what kind of functions must be provided by the product and how functions compose and interact (functional decomposition and functional constraints). Key components take on the main responsibility in fulfilling a function, additional components are then included in a second step. This restriction can be used to simplify the mapping from function to component hierarchies.

Research revolves around modeling and configuring a single system family or product line. However, we expect tradeoffs to arise when multiple system families have to be configured which share commonalities. Also the mapping of functional to structural knowledge is not explicitly formalized with exception of [Soininen *et al.*, 1998], who propose implementation constraints to describe the mapping. We therefore propose to model a functional representation of a technical system and to map this in a second step to a component-oriented model that can be manufactured.

3 Modeling Multiple System Families

The general modeling concepts used to represent a product model are somewhat similar to Object-Oriented principles in software engineering or to the modeling concepts proposed in [Soininen *et al.*, 1998]. For this reason we use UML to represent the model. Figure 3 shows the main concepts used in modeling a product structure. The organization is shown as a part-of structure of function and component types respectively. Function and component types can have attributes which we omit for simplicity. The mapping between functional and component elements is shown using the realizes relation. And an instance of a component class uses the instantiates relationship. Generalized types use the inheritance relationship.

3.1 Defining a model for a single system family

The functional representation of a bridge is shown in Figure 5. We use the part-of relationship to express functional

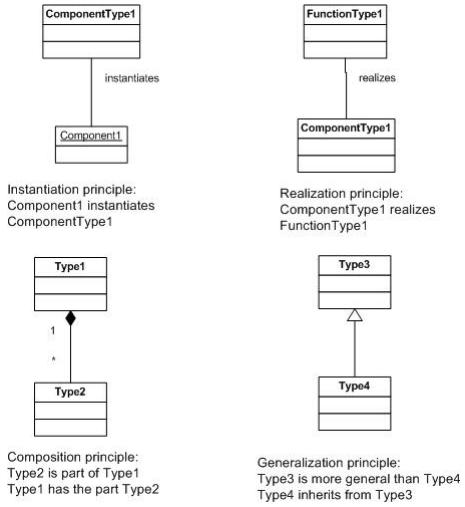


Figure 3: *Part-of, generalization, realization and instantiation relations between function and component types.*

decomposition. A bridge satisfies two functions: it provides a way to cross an obstacle - a valley, a river, another road - and it has to support a given load, e.g. pedestrians walking over the bridge or cars driving on it.

A specific type of bridge is the beam bridge. It is by far the most common one used in crossing over other driveways. We can see in Figure 2 that a beam bridge consists of an upper structure and a lower structure. The lower structure provides support with its piers and the upper structure provides the pathway using a deck placed on beams. The functional representation of the technical system "beam bridge" is immediate (Figure 5). In this case it is easy to establish a model that accommodates for functional requirements. A mapping towards system-line specific components is done in a second step using a similar component-oriented model. As an example we could continue to specify specific beam variants as shown in Figure 4 for each conceptual beam in Figure 5. These beam variants would then need to be mapped to manufacturable parts in a specific configuration. A conceptual beam could be further specified to be an I-beam. Each I-beam would need to be mapped to a set of I-beam units of equal maximal length that can be manufactured. The resulting mapping is shown in 6. Often components are manufactured or ordered based on a specification which can be drawn from an ERP system. In this case it makes sense to use the result of component instantiation in the leaf of the representation tree for indexing the ERP to retrieve a complete specification for a single instance, i.e. to create the final bill of material.

3.2 Extending a Model to Multiple System Families

If we would like to integrate further system families into a configuration in order to automate their respective offerings we might have to extend the initial product model. A way to proceed with an extension to the next n system families can be summarized as follows:

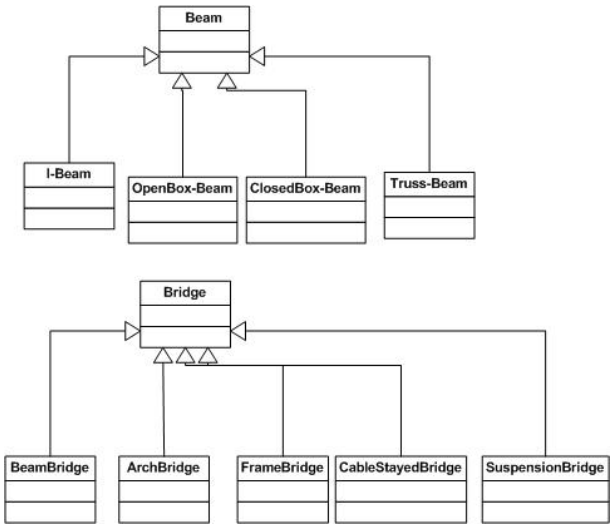


Figure 4: *Example of bridge and beam types.*

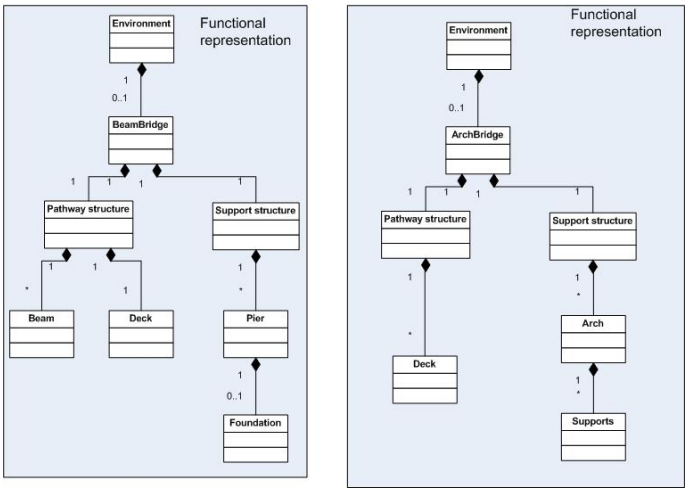


Figure 5: *A functional representation of a beam bridge (to the left) and of an arch bridge (to the right).*

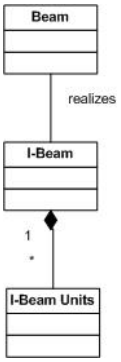


Figure 6: *The mapping from the functional representation to the product oriented model is shown with the realizes relationship. A concrete variant the specification of which might be retrieved from an ERP system is then shown by the instantiate relationship.*

1. Model the functional representation of the individual system families.
2. Encapsulate generic concepts as functions of all system families
3. Use multiplicities, options in part-of relationships as well as the addition of new types in the inheritance relationship in order to model variation.
4. Use additional constraints and rules to select the corresponding product model
5. Define a level in the functional hierarchy with which all families share the hierarchy and from which on they develop their own hierarchy. Two extreme cases are the superset and the intersection functional product model.
6. Define component-oriented structure for the specific system families.
7. Define function-to-component mapping using constraints.

The functional representation of the system family for an arch bridge is similar to the one of the beam bridge (Figure 5). In this case a common functional product model across both system families could be the one shown in Figure 7. Mainly, the generic functional product model in this example is a superset of all choices for all system families. We call it the *superset functional product model*. A set of beams is part of the PathwayStructure only for the beam bridge. It is also part of the PathwayStructure in the generic functional product model. The part-of relation now indicates a multiplicity of 0..n which means that in the generic model there might be no beams. Similarly we handle the part-of relation between the support structure and the pier and the arch. Only the deck function is always available. In other words, the superset functional product model represents a superset of all possible parts some of them optional so that they can be switched on and off for the corresponding system family. The attentive reader may also note that we only need to add an optional cable class as a part of a pier in order to accommodate for cable-stayed bridges. Similarly changes can be adopted to include the suspension and the frame bridge into the generic product model. Additional selection rules have to be defined in order to ensure structural consistency and to generate the final functional product model for a specific system family. In our case additional rules would be the ones shown in Table 1.

Another way to reuse existing knowledge is to define the intersection of all parts in the functional product model as the generic functional product model. In the example this results in two functions for a bridge: the PathwayStructure and the SupportStructure. A selection for a specific family would result in an addition of the system family specific parts using for example inheritance(Figure 5). We call this model *intersection functional product model*.

Observe that in our example a specific bridge type can be selected based on characteristics of the environment. Typical rules are summarized in Table 2. They are extracted as common knowledge from the observation of how bridge designers work [Boulanger *et al.*, 1995]. In the representation of a bridge system family the environment is the top level object (Figure 7). This way the selection process for a specific

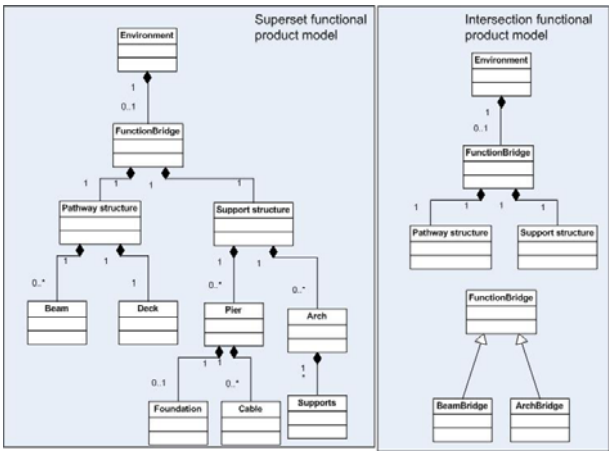


Figure 7: *Generic functional representation using the superset (to the left) and the intersection functional product model (to the right).*

if	bridge type is beam bridge
then	pathway structure has at least one beam
and	support structure has no arch
and	support structure has at least one pier
and	pier has no cable
if	bridge type is arch bridge
then	pathway structure has no beam
and	support structure has at least one arch
and	support structure has no pier
if	bridge type is cable-stayed bridge
then	pathway structure has at least one beam
and	support structure has no arch
and	support structure has at least one pier
and	pier has at least one cable

Table 1: Rules for selecting a system family, in this example the product model for a given bridge type.

if	environment is V-shape
and	environment height is moderate
and	environment length is moderate
then	possible bridge type is beam bridge
if	environment symmetry is not smmetrical
then	possible bridge type is not frame bridge
if	environment height is low or moderate
then	possible bridge type is arch bridge
if	obstacle size is very large
or	environment length is moderate, long or very long
then	possible bridge type is cable bridge

Table 2: Rules for selecting a bridge type.

bridge type can be forced in the initial stage of the configuration based on a few input parameters describing the environment and explicit selection constraints. In other cases a part of the system might have to be generated before a decision on a specific system type can be reached. If the environment is for example not symmetrical, all besides the frame bridge type are possible. In this case, a configuration task has to loop over the different bridge types to find possible solutions. If the initial input parameters completely determine a system family, its selection might occur in the beginning of the configuration process. In this case the selection can be viewed as a configuration task that provides as a result the model of the chosen system family. The navigation between product selection and product refinement has to be well-conceived. If some of the parameters that determine a system family are also relevant for the internal structure, it should only be possible to change them in the range of values allowed for the specified system family. As an example if the maximal load for a beam bridge type is specified it should not be possible to arbitrarily remove piers in a later phase already committed to the beam bridge type in order not to compromise on stability.

Until now we have only concentrated on the product modeling aspects for one or multiple product lines or system families. It might also make sense to evaluate a generic functional product model in terms of usage and maintenance. A product model is used in the configurator to generate a legal i.e. a consistent instance of a technical system, also called a configuration. The individual components in a configuration are instances of the classes we have shown in the product model. In other words each instance refers to its class in terms of a common representation of characteristics and of relationships to neighbor components. In the case of the superset functional product model there is no longer a one-to-one relationship between the structure of the individual configuration and the generic product model. In order to ensure that a given configuration has been generated from such a generic product model one needs to go through a additional rule checks (the ones shown in Table 1). From maintenance point of view a change in the product model of such a superset, for example by adding or removing a part, may require the regeneration of all existing configurations. In the worst case, if the change happens higher up in the part-of hierarchy all system families are affected. In the least a careful analysis would need to determine which ones are affected. In the second case of an

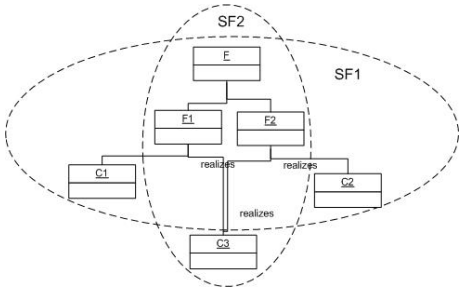


Figure 8: *A multifunctional component C3 realizing two functions F1 and F2.*

intersection functional product model, the specific structure of the system family is generated as an add-on to the generic structure. Thus a clear interface between the common and the individual parts is established. From a usage point of view an individual configuration will have to be checked with respect to the generic part that is common to all configurations and with respect to the product model of the chosen system family. From a maintenance point of view, as long as the common part is not touched, changes in the model of the system family will only have an effect on the existing configurations of that system family. Obviously the reuse factor is lower in this case, which means that distributed development and maintenance would be possible in this setting whereas the superset functional product model would require an authority that closely monitors changes in the generic product model.

4 Function-Structure Mapping

Another difficulty which (un)fortunately does not occur in the bridge design example is the modeling of multi-function components. Multi-function components realize more than one function. It may happen that in the system family SF1 function F1 is realized by a component C1 and function F2 by component C2. In a more modern family SF2 the designers create a component C3 which encompasses F1 and F2 and which is designed in a way to keep F2 optional. In other words component C3 can either satisfy F1 or F1 and F2. In this case the mapping between functions and components is n-to-n (Figure 8). [Mittal and Frayman, 1989] propose to implement such multi-functional components by introducing the concept of key components. In Figure 8, C1 and C3 would be the key components for F. If C1 would be chosen to realize F, C2 had to be added additionally. The functional decomposition of F into F1 and F2 is bypassed through the notion of key components. In our bridge example the use of such key components bypassing the functional decomposition would be somewhat counterintuitive as the justification for choosing a functional representation lies in its stable nature as well as the capturing of customer input. Also, as mentioned in [Mittal and Frayman, 1989], using key components starts to blur the distinction between functions and components. And one use of the functional hierarchy is to make explicit system dependencies on a functional level. Our proposal is to keep the decomposition of F into F1 and F2 in the model. The information whether an additional component C2 would have to be needed to satisfy F2 is included via implementation constraints [Soininen *et al.*, 1998] based on the specific system family. If system family SF1 is chosen C3 is generated for F1 and F2 points to C3 as well. For SF2, F1 would require C1 and F2 realized by C2. In some cases this formulation may also be independent of any system family selection and only depend on the mapping between function and components.

5 Conclusions

We have discussed in this paper product models for technical systems that are configured. The challenge is to carefully define a model that is as stable as possible over several system families. For this reason we propose a functional model of the technical system and we discuss the advantages and disadvantages in use and maintenance of two types of generic product models across multiple system families: the superset and the intersection product model. In the future we will concentrate on further modeling issues such as the multi-functional component briefly presented in the end.

References

[Boulanger *et al.*, 1995] S. Boulanger, E. Gelle, and I. Smith. Taking advantage of design process models. In *IABSE Colloquium*, March 1995.

[Göpfert and Steinbrecher, 2000] Jan Göpfert and Michael Steinbrecher. Modulare produktentwicklung leistet mehr. *Harvard Business Manager*, Heft 3:20–30, 2000.

[Mittal and Frayman, 1989] Sanjay Mittal and Felix Frayman. Towards a generic model of configuration tasks. In *Proc. 11th Int. Joint Conf. on Artificial Intelligence (IJCAI-89)*, pages 1395–1401, 1989.

[Najman and Stein, 1992] O. Najman and B. Stein. A theoretical framework for configurations. *In Proceedings of Industrial and engineering applications of artifical intelligence and expert systems, IEA/AIE-92*, pages 441–450, 1992.

[Pernler and Leitgeb, 1996] Staffan Pernler and Max Leitgeb. Functional and structural reasoning in configuration tasks. *AAAI-96 Fall Symposium Series*, pages 119–123, 1996.

[Salustri, 1996] Filippo A. Salustri. A formal theory for knowledge-based product model representation. *2nd IFIP Knowledge Intensive CAD Workshop*, 1996.

[Shai and Preiss, 1999] O. Shai and K. Preiss. Graph theory representations of engineering systems and their embedded knowledge. *Artificial Intelligence in Engineering*, 33:273–285, 1999.

[Simon, 1981] Herbert A. Simon. *The Sciences of the Artificial*. MIT Press, Cambridge, Massachusetts, second edition, 1981.

[Soininen *et al.*, 1998] Timo Soininen, Juha Tiihonen, Tomi Männistö, and Reijo Sulonen. Towards a general ontology of configuration. Technical Report TKO-B141, Laboratory of Information Processing, Helsinki University of Technology (HUT), Helsinki, Finland, 1998.

Design Space Exploration Using Constraint Satisfaction

Noel Titus*, Karthik Ramani
Purdue Research and Education Center for Information Systems in Engineering (PRECISE)
School of Mechanical Engineering, Purdue University,
585 Purdue Mall, Room 1, West Lafayette, IN 47907, USA

Abstract

In this paper, enablers necessary for design exploration in a product realization environment are discussed. A product representation schema is suggested for constraint satisfaction based search and exploration of the design space for customized products. A formulation of concept design as a CSP is presented. The importance of analysis and simulation of systems during concept design in evaluating the design space, from both a feasibility and optimality stand point, is examined. Capabilities required of CSP algorithms for engineering based design configuration are listed. Finally, a preliminary case study of an industrial erector set configurator, implemented in ILOG, is presented to exhibit the some of the issues stated above.

Keywords

Constraint satisfaction, design and configuration, knowledge representation

1 Introduction

Current trends in product development center around delivering products that meet user requirements as closely as possible, achieving concurrency in the development stage between all entities involved in the product life cycle and enabling product realization in a quick manner. These goals must be achieved in a distributed setting, since the product development centers, manufacturing units and retailers are knowledge centers that are dispersed geographically. The motivation for attaining these goals originates from the fact that decisions made during the conceptual and design phases account for 80% or more of the cost of the product [Dieter99].

This paper discusses developments required in order to achieve the above goals using constraint satisfaction. The outline of the paper is as follows:

Section two presents the concept for a representation schema for products that encompasses geometry, manufacturing, and logistical variables and the mappings between the variables. Such a product centric representation of the lifecycle variables will be used in a distributed constraint satisfaction and optimization based design exploration framework. In section three, a formulation of concept design as a CSP is presented. Capabilities required of CSP algorithms to enable product configuration and synthesis using constraint satisfaction are also examined. An implementation of a simple product configurator for an industrial erector set is given in section four, to help put some of the issues discussed in the prior sections into perspective.

2 Product Representation Schema for CSP-based Design Space Exploration

In the product lifecycle (Fig. 1(a)), first, the customer requirements (CRs) are identified, and mapped to engineering requirements using techniques such as Quality Function Deployment (QFD). Concepts are then generated from the functional decomposition to meet the engineering requirements (ERs). These concepts are represented by design variables or parameters, which when associated with values, impart geometrical form, performance characteristics and qualitative aspects to the concept design.

In [Ashby99] it is shown that function, shape, material and process are interrelated. This implies that the entities are not independent. If one of the four entities involved is fixed, it constrains the remaining three entities to values belonging to a subset of their complete domains. As a result, the decisions made in selecting values for the design parameters during conceptual design have a constraining effect on selectable processes and materials. In fact, the design decisions made have a far reaching effect, extending through the lifecycle of a product.

Consider a design problem where the customer requirement is to obtain speed reduction from a motor to drive a load shaft. The progression of the solution to this problem through its lifecycle is shown in Fig. 1(b). The customer requirements drive the engineering requirements and conceptual phases. A helical gear [Shigley01] assembly is chosen as the concept. Selected relationships between the different phases can be represented as follows:

* Corresponding author: ntitus@purdue.edu

- Manufacturing phase variables
  Process P = F(Gear form, Tolerance, Material)
  Supplier S = G(CRs, Lead time, Capacity, Process)
- Operation phase variables
  Reliability R = H(ERs, Material, Process)

Here F, G and H represent functions. The functions are mappings that are either discrete one-to-one, one-to-many or many-to-many mappings or are continuous functions with definite domains. The product representation schema should capture these variables and mappings.

Standards for product representation are outlined in [Wong93], [Szykman01], and [Xue04]. The representation outlined in these papers goes beyond representing the geometry of the design artifact, to include function, behavior, physical and functional decomposition. A generic product information core was developed, that would serve as a standard to enable interoperability, concurrent development etc. However, the representation contains only the taxonomy of artifacts and their function, form and relationships. As a result, it is limited in application.

In [Gero90] a design prototype is presented symbolically as P = (F, B, S, D, K, C); where F represents function, B represents behavior, S represents structure, D represents design description, K represents knowledge, C represents context. The key point in the above design prototype is that knowledge (K) comprises of the following: Relational knowledge which establishes dependencies between F, S and B; qualitative knowledge which contains the effects of modifying the variables of S on B and F; computational knowledge that specifies symbolic or mathematical relationships; constraints which appear as expected behavior; and context knowledge. Such a representation is suited to configuration and synthesis in the product design phase.

In order to realize products while taking into account lifecycle issues, variables from different phases of the product lifecycle such as manufacturing, distribution, operation and retirement should also be considered. We propose augmenting the above representation of products to encompass these variables and their mappings. Capturing the variables in each phase of the product lifecycle and the mappings between the variables will allow for exploration of the design space using CSP techniques. The CSP algorithm will evaluate all the constraints represented by the mappings for violation, during feasibility evaluation.

Combining this product representation schema with a standardized method of representing a technical system and its decomposition into sub-systems [Gelle04] will provide a common interface in data exchange and reuse scenarios.

3 Formulation of the concept design as a CSP

This section is divided into three sub-sections. The first sub-section reviews optimization techniques that utilize model analysis for concept design. The second sub-section addresses the potential role of CSP in the concept design. The capabilities required of CSPs for this purpose are also discussed. Finally, the third sub-section proposes a formulation of a design concept as a CSP. This formulation is explained using a representation called "concept graph".

3.1 Prior work in analysis-based concept design synthesis

Consider the vehicle decomposition in [Michelena99], as seen in figure 2. The vehicle is hierarchically decomposed into systems, sub-systems and components. Note that such decomposition is possible only once specific concepts have been selected for the functions in the functional decomposition. The vehicle requirements are translated into vehicle design targets for the systems, which in turn are cascaded down to subsystem and component targets. This translation is accomplished by means of analytical models of the subsystems and sensitivity analysis. Analysis is necessary to translate customer requirements to engineering requirements for systems, sub-systems and components.

In the example, once the targets are established for the nodes in the decomposition, analytical target cascading (ATC) is used to determine optimum values for the design parameters for all nodes. Since the systems represented by the nodes are uncoupled this is done in parallel. Analysis of the node models are used to determine whether a target is

(a)

DETERMINE CUSTOMER REQUIREMENTS (CR)	CONCEPTUAL PHASE	DESIGN PHASE	MANUFACTURING PHASE	DISTRIBUTION PHASE	OPERATION/ USE PHASE	RETIREMENT PHASE
	• Generate engineering requirements (ER) • Functional decomposition • Generate concepts	• Design variables instantiation • Material, form decisions • Performance analysis	• Process decision • Supplier decision • Manufacturability issues	• Warehousing, retailer decisions • Market decisions	• Maintenance, repair issues • Reliability • Functioning environment	• Environment issues • Disposal issues • Regulatory issues

(b)

CUSTOMER REQUIREMENTS	CONCEPTUAL PHASE	DESIGN PHASE	MANUFACTURING PHASE	DISTRIBUTION PHASE	OPERATION/ USE PHASE	RETIREMENT PHASE
• Operate low speed machine from high speed motor • Cost < c • Volume < v • Weight < w • Life > l • Delivery time < t	• ERs: Reduction ratio x:1, strength, cost, noise, wear resistance, weight • Belt drives, gear assembly →Gear assembly selected	• Gear form: Assign pitch, # of teeth, pressure angle, tolerance etc. • Material: Cast Al, steel → steel • Force, strength analysis	• Processes: Die casting, gear hobbing, milling • Supplier decision: lead time, capacity	• Transport to customer	• Reliability: Load cycles = N	• Steel recycling • Dismantling

Fig. 1: (a) Product Lifecycle (b) Product Lifecycle for a gear assembly

met. The goals of Analytical Target Cascading (ATC) are to enable the design of complex systems to occur at a much earlier stage and to generate prototypes that more effectively satisfy functional requirements.

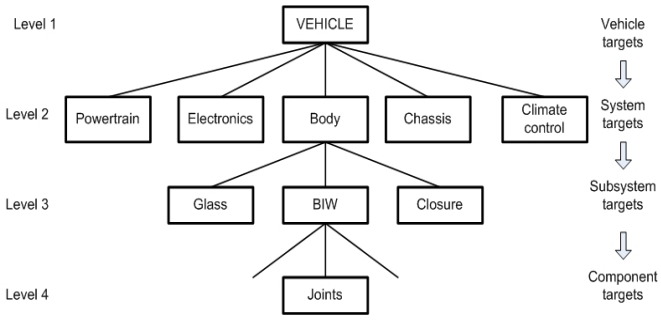


Fig. 2: Decomposition of automobile into systems

A detailed description of ATC is available in [Allison04]. In such a problem, if the design parameters have continuous domains, gradient based approaches can be used to obtain parameter values. If the domains of selectable values are discrete, gradient free algorithms such as Genetic Algorithms (GA) can be utilized. For example, GAs were used to obtain instantiations for joint types and their optimal location in a structural frame for the "joints" node in Fig.2 [Saitou04].

As seen above, analysis combined with a hierarchical optimization scheme or stochastic optimization can solve a concept design problem. The next few paragraphs attempt to make a preliminary evaluation or the role of CSPs in design. A comparison of stochastic and systematic constraint satisfaction is available in [Freuder95].

3.2 Role and requirements of CSP in concept design

In the discrete optimization problem, Fig. 2, only feasible joints were included in the domain, though a creative formulation of an objective function can help determine feasibility in case infeasible joints are present in the domain. Using CSP, it is possible to determine whether a design problem is satisfiable, i.e. feasible.

CSP solution methodology should be harnessed for its ability to produce all possible valid solutions, which is helpful in a design exploration scenario. When optimality considerations are important, branch and bound search techniques for optimization in CSPs, such as those in [Larossa98] can be used. GAs have been applied to constraint satisfaction optimization problems in [Tsang90].

The ATC problem described is uncoupled allowing optimization of the different nodes to run in parallel. In the circumstance that there is a coupled problem, with domain values that are potentially infeasible, CSPs can play a part. This will call for constraint satisfaction between two modules that are being synthesized using optimization or CSP techniques. Such a coupling is captured in Fig. 3. From the discussion in section 3.1, it is clear analysis plays a key role in design evaluation. CSP algorithms should be able to handle constraints, the satisfaction or violation of which depends on the results of an analysis of models representing a system, such as those that appear in Fig. 2.

In addition, the following combined capabilities (1 – 4) are required of CSPs in a design synthesis environment:

1. Continuous and discrete domain handling: Design parameter domains are both continuous and discrete. CSP should be able to handle both continuous and discrete domains. Gelle has presented formalism for conditional mixed constraint CSPs in [Gelle03].

2. Dynamic constraint handling: During the design process, the product design specifications evolve and change constantly. The design specifications, which represent constraints, may be changed to include new variables i.e. new constraints may be inserted, or they may be relaxed or removed. CSP algorithms need to be able to handle this dynamicity of constraints. Dynamic constraint satisfaction algorithms have been described in [Mittal90] and [Soininen99]. A partial constraint satisfaction algorithm useful in such a situation was suggested first in [Freuder89].

3. Distributed constraint handling: In the current business environment design and configuration activities occur in a distributed manner. Yokoo and Hirayama have developed multi-agent system based algorithms for distributed constraint satisfaction [Yokoo00]. Felfernig *et. al.* [Felfernig01] formalize a definition of distributed constraint satisfaction, and presents a basic architecture and cooperation algorithm for distributed configuration systems.

4. Object oriented constraint satisfaction: It is useful to draw parallels between the physical world and its software representation. The object oriented paradigm will help in capturing this translation from physical to virtual. Work in object oriented constraint programming is described in [Roy97]. Mailharro presents a framework for configuration, the architecture of which is based on [Stumptner98], that uses CSP and object oriented paradigms in [Mailharro98].

3.3 Concept design as a CSP

Any product that is an assembly can be represented as a graph structure. In its simplest form, the nodes of the graph are the sub-assemblies or atomic components in the assembly and the arcs represent the physical connectivity between the sub-assemblies or components. When used in this manner, the graph structure captures the topology of the product and is called a "concept graph" Such a graph structure of a coupled assembly structure is in shown in Fig.3. Each of the nodes could be any one of the "concepts" from a given level from the decomposition in Fig. 2. In essence, every node is a system in the overall concept. For example, in an automobile the nodes of the graph could be power train (S1), electronics (S2), chassis (S3), and suspension system (S4). The nodes in the concept graph will be represented by an object that encapsulates the variables included in the representation schema outlined in section 1. In a CSP framework, the nodes in the graph constitute the variables. The domains of the nodes are all possible instantiations for the systems in the concept i.e. instantiations of S1, S2, S3 and S4.

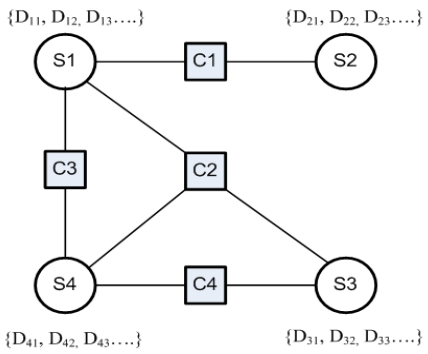


Fig. 3: Concept graph

In Fig. 3, the domain values are denoted by $D_{xy,}$ where x is the index of the system S_x, and y is the index of the particular instantiation in the domain. These domains are generated by optimization based synthesis, catalog based configuration, simple table lookups etc. The arcs in the concept graph that connect the nodes are constraints between the systems of the overall concept. These constraints between the nodes are the mappings between variables of the systems, which is covered in section 2. The constraints can be unary or higher order constraints. In Fig. 3 the constraints are indicated by C1, C2, C3 and C4. Hence, a CSP representation of a concept design consists of variables S1, S2, S3,…, Sn, which are systems, subsystems, or components, whose values are taken from domains D1, D2, D3,…, Dn. The constraints Ck are the mappings or geometrical constraints etc. between the system variables. A point to note is that the concept graph has a recursive definition, because each system in the graph can be split into sub-systems until the atomic level of components is reached.

4 Product configurator for an industrial erector set

In order to understand issues related to configuration using CSP from an engineering standpoint, work on an industrial erector set (IES) configurator was initiated. The IES catalog consists of sixteen extrusions of different cross sections and one hundred and four joining plates. The erector set also consists of dynamic pivots, linear motion bearings etc. The extrusions are connected using joining plates or fasteners to form any kind of frame structure. Applications of the IES include machine frames to mount machines and tooling, furniture, and machine guards. In this configurator problem, only extrusions and joining plates were considered. A system for selection of adequate wind bracing for steel framed buildings using local consistency and conditional constraint satisfaction is presented in [Gelle00].
The final goal is to generate all possible configurations of valid extrusion and joining plates from the IES catalog based on the user's desired frame structure, which is specified by spatial location of frame members, load on the frame and personal preferences.

4.1 System architecture

The configurator architecture is shown in Fig. 4. The components are explained below; a (I) or (NI) after the module indicates that the module has been implemented or not been implemented respectively.
User interface (I): The user interface has two parts: a display pane where the user enters the coordinates of the frame structure, the load and load direction and a visual rendering of the frame is displayed; and a user preference pane where unary preference constraints for extrusions are stated.
Analysis module (NI): The analysis module is used to determine the transverse, axial, bending moment, and torsional loads on the frame which translate to strength constraints on extrusions and joining plates. One way to implement this module is to submit the spatial and load information of the frame to a finite element analysis (FEA) solver. Currently, the user does the static analysis. Analysis is required to convert the system or customer requirements to engineering requirements for each frame component.
Query Interpreter (I): The query interpreter uses rules to insert a joining plate between every connected horizontal and vertical extrusion and outputs a graph structure containing spatial and loading information of the frame structure.

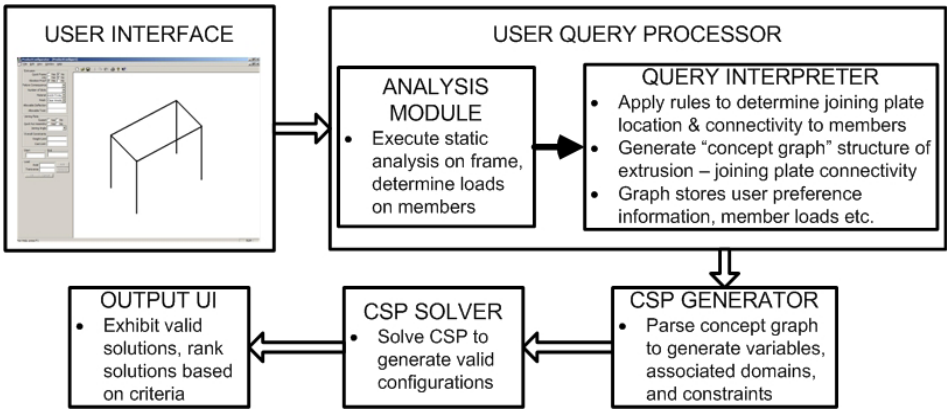


Fig. 4: System architecture for an Industrial Erector Set Configurator

CSP Generator (I): The CSP generator parses the adjacency linked-list of the concept graph and passes the variables, constraints and domains to the APIs of the CSP solver.
CSP Solver (I): The CSP Solver has been implemented using ILOG Solver.
Output UI (NI): The output user interface (UI) will exhibit the results from the CSP solver. It will rank the results based on user specified criteria such as cost, weight etc.
This system is capable of generating a CSP on the fly for every distinct query. It does not require "precompiled" solution spaces, where all possible solutions are enumerated. Since each query will involve a different number of variables, such an enumeration of the valid solution space will not work.

4.2 Sample problem

The IES product configurator was queried with a simple table frame structure, with four vertical members supporting four horizontal members each with a transverse load of 100lbs. The query and concept graph of this problem is indicated in the Fig 5 (a), (b). The CSP formulation is:
Variables (V): E – extrusion, J – joining plate
$V\{E_1, E_2, E_3, E_4, E_5, E_6, E_7, E_8, J_1, J_2, J_3, J_4, J_5, J_6, J_7, J_8\}$
Domains (D):
Extrusions - $D_E\{16 \text{ extrusions}\}$,
Joining plates – $D_J\{104 \text{ joining plates}\}$
Examples of extrusions and joining plates are in Fig.5 (c).
Constraints (C):
The constraints are summarized in table 1.

4.3 Results

The IES configurator system was queried with the above table type frame structure with a transverse load of 100 lbs on the horizontal members. The user placed a constraint that required all extrusions and joining plates used in the frame to be the same. The user preference query to the system for the table structure and corresponding number of valid configurations output is summarized in table 2. Other combinations of user preferences did not generate valid configurations, i.e. the CSP was not solvable.
A CAD model of a table frame assembly using two slotted 2"x2" square extrusion using corner brackets of 2" width is shown in figure 6. This solution is one of the valid configurations generated in query 3 in table 2.

Unary Constraints: ***Preference and Structural constraints***	
Attribute(Domain)	***Comment***
Extrusion.quickFrame = (Y/ N)	Quick frame type extrusions allow for quick assembly of frames
Extrusion.vibrationProof = (Y/ N)	Vibration proof extrusions do not loosen under vibratory loads
Extrusion.loadApplication = (0 / 1/ 2)	1: Low – static loads, 2: Medium – some varying loads, 3: High – impact loading possible
Extrusion.MI >= required MI	Extrusion moment of inertia (MI) given in catalog
JoiningPlate.(DL/BL/SL) Capacity>=applied(DL/ BL/ SL)	Joining plate direct (DL), bending (BL) and shear (SL) load capacities given in catalog
Binary constraints: ***Mating constraints***	
(Ext1.L = JP.R & Ext2.L = JP.C) || (Ext1.B = JP.R & Ext2.B = JP.C) || (Ext1.L = JP.C & Ext2.L = JP.R) || (Ext1.B = JP.C & Ext2.B = JP.R) || (Ext1.L = JP.R & Ext2.L = JP.C) || (Ext1.B = JP.R & Ext2.B = JP.C) || (Ext1.L = JP.C & Ext2.L = JP.R) || (Ext1.B = JP.C & Ext2.B = JP.R) *where L – number of slots along length, B – number of slots along breadth, R – number of rows of holes, C – number of columns of holes and Ext 1 and Ext 2 are two extrusions connected to joining plate JP*	

Table 1: Summary of unary and binary constraints

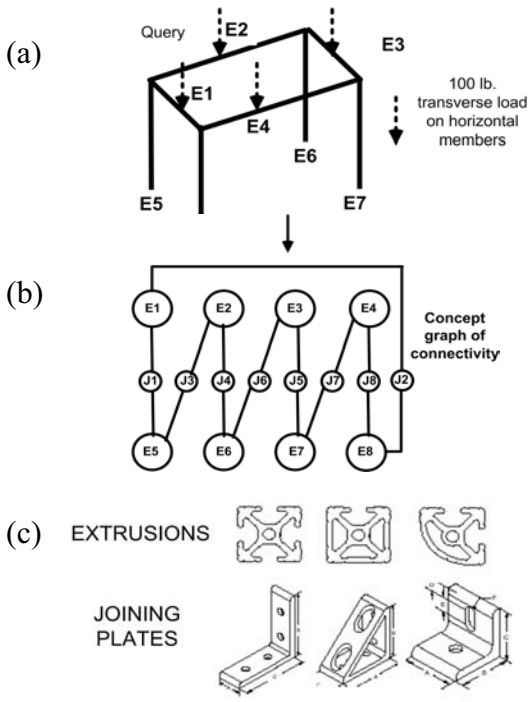


Fig. 5: (a) User query to configurator
(b) Concept graph of user query
(c) Examples of extrusions and joining plates

Query number	Quick Frame	Vibration proof	Lite	Load App.	Number of valid configs.
1	No	Yes	No	Low	48
2	No	Yes	No	Med	50
3	No	Yes	No	High	122
4	No	Yes	Yes	Low	44
5	No	Yes	Yes	Med	34

Table 2: User preference query and solutions generated

5 Future work

The product representation schema discussed in section two will be developed and then tested in the IES configurator framework and through complex examples such as automobile sub-systems. This representation will then be used in the CSP framework (section three) to evaluate for valid configurations that meet performance and functional requirements. The IES system architecture will be implemented in its entirety, which will allow the system user to get valid CAD assembly models from a submitted query. Additional rules and optimization modules will also be developed.

Fig. 6: Table frame CAD model

Acknowledgements

We would like to thank Rajasekar Karthik for his help in the implementation of the IES product configurator.

References

[Allison04] Allison, J.T., Complex System Optimization: A Review of Analytical Target Cascading, Collaborative Optimization, and Other Formulations. *Master's thesis, University of Michigan,* 2004.

[Ashby99] Ashby, M. F. *Materials Selection in Mechanical Design*, Second ed., Butterworth Heinemann, Oxford, UK. 1999

[Dieter99] Dieter, G. E. *Engineering Design: A Materials and Processing Approach*, Third ed., McGraw-Hill, New York, USA. 1999.

[Felfernig01] Felfernig A., Friedrich G., Jannach D., Zanker M., Towards Distributed Configuration, *Proc. 24th German/ 9th Austrian Conference on Artificial Intelligence*, 918 – 212, 2001

[Freuder89] Freuder, E. C., Wallace, R. J. Partial Constraint Satisfaction. *Proc. of the Eleventh International Joint Conference on Artificial Intelligence*, IJCAI-89: 278-283, 1989.

[Freuder95] Freuder, E., Dechter, R., Selman, B., Ginsberg, M., and Tsang, E., Systematic Versus Stochastic Constraint Satisfaction, Panel, *IJCAI-95*:2027-2032, 1995.

[Gelle00] Gelle, E., Faltings, B., Smith, I., Structural engineering design support by constraint satisfaction. *Artificial Intelligence in Design*: 311 – 331, 2000.

[Gelle03] Gelle, E., Faltings, B., Solving mixed and conditional constraint satisfaction problems, *Constraints*, 8:107 – 141, 2003

[Gelle04] Gelle, E.,, Consistency Rules for Modelling Technical Systems based on IEC 61346, *Configuration Workshop at ECAI 2004*, Valencia, August 2004.

[Gero90] Gero, J.S. Design prototypes: a knowledge representation schema for design. *AI Magazine*, 11(4): 26-48, 1990.

[Larossa98] Larossa, J. Algorithms and heuristics for total and partial constraint satisfaction, *PhD dissertation, Universitat Politecnica de Catalunya*, 1998

[Mailharro98] Mailharro, D., A classification and constraint-based framework for configuration, *AIEDAM*, 12: 383 – 397, 1998.

[Michelena99] Michelena, N., Kim, H. M., Papalambros, P. A system partitioning and optimization approach to target cascading. *International Conference on Engineering Design*, ICED-99, Munich, 1999

[Mittal90] Mittal, S., Falkenhainer, B., Dynamic constraint satisfaction problems, *Proceedings of the 8th National Conference on Artificial Intelligence*: 25-32, 1990.

[Roy97] Roy, P., Liret, A., Pachet, F., A framework for object-oriented constraint satisfaction problems, *Object-Oriented Frameworks*, ACM Press, Eds. Fayad, M., Schmidt, 1997.

[Saitou04] Saitou, K., Lyu, N. Decomposition-based assembly synthesis of space frame structures using joint library, *Proceedings of DETC'04*, DETC-04, Salt Lake City, UT, 2004.

[Shigley01] Shigley, J. E., Mischke, C. R., Mechanical Engineering Design, 6ed., *McGraw-Hill*, 2001

[Stumptner98] Stumptner, M., Friedrich, G., Haselboeck, A., Generative Constraint-Based Configuration of Large Technical Systems, *Special Issue on Configuration: Artificial Intelligence in Engineering, Design, Analysis and Manufacturing*, 12(4), December 1998

[Soininen99] Soininen, T., Gelle, E., Niemela, I., A Fixpoint Definition of Dynamic Constraint Satisfaction, *5th International Conference on Principles and Practice of Constraints Programming – CP99*: 419-433, Alexandria, VA, 1999.

[Szykman01] Szykman, S., Fenves, S. J., Keirouz, W., Shooter, S. B. A foundation for interoperability in next-generation product development systems. *CAD*, 33:545 – 559, 2001

[Tsang90]Tsang, E. P. K., Warwick, T., Applying genetic algorithms to constraint satisfaction problems, *Proceedings European Conference on AI*, ECAI90: 649 – 654, 1990.

[Wong93] Wong, A., Sriram, R. D. SHARED: an information model for cooperative product development. *Research in Engineering Design*, 5(1):21 – 39, 1993

[Xue04] Xue, D., Yang, H. A concurrent engineering-oriented design database representation model. *CAD*, 36: 947 – 965, 2004.

[Yokoo00] Yokoo, M., Hirayama, K., Algorithms for Distributed Constraint Satisfaction: A Review, *Autonomous Agents and Multi-agent Systems*, 3(2): 198 – 212, 2000.

Configuring Loopholes: Interactive Consistency Maintenance of Business Rules

Markus Stumptner, Michael Schrefl[1]

Abstract. Business rules are the core of information systems. They define under which conditions certain actions may, must not, or need to be taken. Business rules are often collected as such during analysis, but are frequently buried during implementation in different form of application code (within procedures, as pre- or postconditions, or as database triggers). Part of the conceptual design process is to make business rules explicit and associate them with activities of object types.

Object-oriented design methods, such as UML, organise object types in pecialisation hierarchies. In the past, various consistency notions have been eveloped for specialising methods (co- and contra-variance), for specialising object-life cycles (observation- and invocation-consistency), and for specialising active rules (database triggers) in object-type hierarchies.

This paper complements these past research results by introducing an application-oriented perspective for the specialisation of business rules. Using the notion of "decision consistency", we introduce a specialization model for business rules that is aligned with commonly used principles in organizational contexts to guarantee consistency between decisions at a higher legislative level (e.g., European Union) and at a lower level (e.g., a member state).

1 Introduction

Business rules are the fundamental tool for decision making. They provide a system of rules that define conditions under which actions may execute. In a company, actions are defined at several levels. As this fact does not provide a comparable consideration of similar initial situations, it is necessary to find a universal/general treatment for decision making.

In this paper we present a modularization of business rules which we call 'decision consistency' such that decisions made on a certain level of authority is comprehended and understood on a higher level of authority. Moreover, this modularization guarantees that decisions made at a higher level are still valid on a lower level. This is what we call decision consistency. We use object/behavior diagrams to model business processes and their associated business rules. Then a business process is an object type and business rules are defined as conditions which properties of a business process have to satisfy. Consider for instance the example of a credit allowance.

Example 1 *A credit institution gives credit to its customers.*

If a person applies for a credit, the creditworthiness of the client has to be checked. For this reason, the case officer handling the credit must collect sufficient information about the client to make a decision.

In addition, different types of credit exist, for example home loans and mortgages, and particular types of credit may be granted by particular divisions in the enterprise, e.g., private loans and mortgages[2] *would be handled by a different division from business loans.*

The natural way of modelling business rules in database applications is by expressing them as ECA (Event-Condition-Action) rules [WC96] - when particular business situations arise, the rules are activated and executed, possibly activating further events. Conceptual database models have long used object-oriented concepts, and object-oriented and object-relational databases associate business rules with particular classes describing domain entities [CCS94] that are integrated into complex business processes. This situation gives rise to the arrangement of rules in inheritance hierarchies, an issue that has been studied (mostly in terms of separately modeling inheritance of events, conditions, and actions) for several years [CMR98, BGM00]. The general principle is that of specialisation (i.e., semantically consistent inheritance) [SS02]: the actions executed in a subclass must be compatible with the actions executed in a superclass under the same event (or a supertype of that event). In the context of business rules, we assume that the specialisation hierarchy of rules follows that of business processes, and the business processes follow the structure of the company, as does the power to make decisions. This is a principle in conceptual rule modeling: An enterprise will typically define a set of rules that describe guidelines that are global to the enterprise (similar to the global schema in the classical 3-Level CODASYL database architecture), and lower level administrative units will have to to function (i.e., define their own rules) within that framework.

This means that particular divisions or functional areas in the enterprise may define their own rules, however these rules have to satisfy two conditions which summarise the notion of *Decision Consistency*:

1. More specific rules (e.g., division-specific rules) *comprise* the enterprise-wide rules, i.e., both division and enterprise specific rules must be covered
2. More specific rules may not contradict the more general rules, i.e., the division specific rules must uphold the principles set up by the encompassing organisation.

[1] Advanced Computing Research Centre, University of South Australia, 5095 Mawson Lakes (Adelaide) SA, mst|cismis@cs.unisa.edu.au

[2] In a mortgage, some asset of the client's is listed as a security in case the client defaults on the loan.

Example 2 *In our example, the enterprise (a bank or other financial institution has general rules about providing loans to customers. Particular divisions that are respnsible for private loans or mortgages will introduce separate business rules that need to be satisfied in their special situations (cf. Figure 1).*

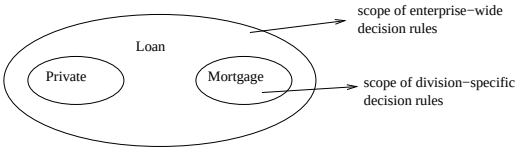


Figure 1. Specialisation of business rules

The formal description of a business rule is as a set of boolean conditions (constraints) over so-called *object situations* $\sigma =< v, l, h, t >$ associated with particular activities in the model. For a given activity a at class O we distinguish the following conditions as definable by a rule:

- Conditions determining *obligations* O^a_o. If an obligation condition is satisfied, action a must be executed.
- Conditions referring to *forbidden actions* F^a_o. If an obligation condition is satisfied, action a must not be executed.
- Conditions referring to *permissions* P^a_o. If a permission condition is satisfied, action a can be executed, but need not be.

We refer to O, F, or P as the *mandate* given by the rule.

It is P^a_o that is required to enable interactive operation: the business rules can explicitly specify openings that are left to the case worker's judgment concerning the choice of particular actions. We add the axiom that O^a_o and F^a_o are mutually contradictory, i.e., it is not acceptable that the same action is prescribed and prohibited at the same time (this is satisfied in the rules depicted in figure 2 and 3). Note also that $\neg F^a_o(\sigma) \rightarrow P^a_o(\sigma)$ does not need to hold.

In the grant example, the three actions available are *grant* (g, the case worker decides that the application succeeds), *reject* (r, the application is unsuccessful), and *forward* (f, the case worker is not authorised to make a decision because this is a borderline case that has to be forward, e.g., to a board of experts that will make the actual decision). The parameters governing the three actions are the loan amount l, the return payment rate R, and the disposable income, d. The coordinate axes are the same for all figures: as specified in the first, they correspond to l and d/R, respectively. The second column of the table below specifies the different action conditions for the general loan category. Figure 2 (right) and 3 show the constraints for each action separately. We refer to the n-dimensional space described by the parameters used in the rule (where n is the number of parameters) as the rule's *decision space*.

The example is a very simple process with a single step; in general earlier actions could be part of the conditions. I.e., essentially a stepwise workflow is described via a set of rules, leading to a larger constraint network.

2 Decision Consistent Specialization

To examine the situation of business rules specialisation, we follow the organisational view described in the previous

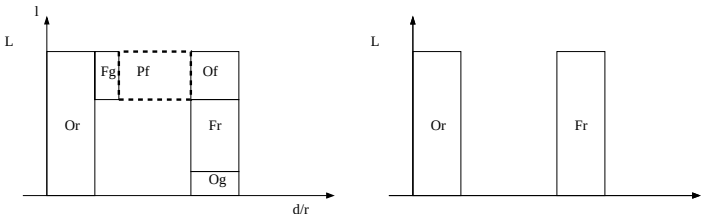


Figure 2. Loan rules: all actions (left); reject (right)

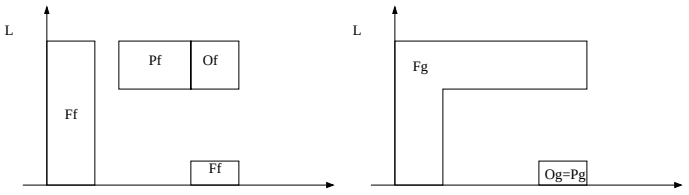


Figure 3. Loan rules: forward (left); grant (right)

section. While the overall organisation has business rules describing the general version of a process, subinits of the organisation have more rules dealing with more specific versions of that process, i.e., their business rules are associated with subunit-specific subclasses of the generic business object.

We use the example of specializing general loan requests to private loan requests. The handling of a request for a private loan differs from a generic loan request insofar as two additional activities are necessary before the requester's credit rating can be evaluated: checking with the credit rating association, and identifying the amount of salary available for impounding if the recipient defaults upon the loan. A mortgage differes from the standard credit rating process by the need to estimate the value of the property being mortgaged, determining the ownership conditions, and identifying the result limit of the mortgage. The correct integration of the new activities and states into existing diagrams depicting the lifecycle of an application object was described in [SS00] for Object/Behavior Diagrams and in [SS02] for UML. Likewise, the specialisation of a set of business rules that refer to a specialized object type has to obey certain conditions that are consistent with the set of rules defined for the original object type.

In particular, the idea is that the set of rules for the subtype should encompass the space of all decisions made by the supertype, but on the other hand, the rules defined for the subtype should not contradict the decisions made for the supertype. In other words, for a given $< Action, Mandate >$ pair, the subtype's rules decision spaces need to be consistent with the decision spaces of the rules in the supertype.

Example 3 *In our banking example, the above informal criteria imply that a private loan must be granted if it must be granted under the rules specified for general loans. In addition, they must not be granted if the rules for a general loan say that it must be rejected. Private loans have as an additional property the impoundable income i.*

For reasons of simplicity we use the same set of actions for the supertype and the subtype; the normal assumption in rules inheritance is that the actions used in the subtype can themselves be specialisations of the actions in the supertype [CMR98, BGM00]. In that case, the subtype actions need to be projected back to their supertype counterparts to test for consistency of the subtype's rule's decision space. This notion has been explored in more detail in [KS95, BS94].

Loan Type/ Activity	Loan	Private Loan
Loan refusal *obligatory* (must)	$i < 0.5R$	$(d < 0.5R)$ or $d < 0.8R$ and $i < R)$
Loan refusal *forbidden*	$d \geq 0.3R$	$(d \geq 3R)$ or $(d \geq 0.8R$ and $i \geq R)$
Passing on to board *obligatory*	$d \geq 0.3R$ and $K \geq 900$	$(d \geq 3R$ and $K \geq 100)$ or $(d \geq 0.8R$ and $i \geq R)$
Passing on to board *obligatory*	$d \geq 0.3R$ and $K \geq 900$	$(d \geq 3R$ and $K \geq 100)$ or $(d \geq 0.8R$ and $i \geq R)$
Passing on to board *forbidden*	$(d < 0.5R)$ or $(d \geq 0.3R$ and $K \geq 100)$	$(d < 0.5R)$ or $(d < 0.8R$ and $i < R)$ or $(d \geq 3R$ and $K < 100)$ or $(d \geq 0.8R$ and $i \geq R$ and $K < 100)$
Loan offer *obligatory*	$d \geq 0.3R$ and $K < 100$	$(d \geq 3R$ and $K < 100)$ or $(d \geq 0.8R$ and $i \geq R$ and $K < 100)$
Loan offer *forbidden*	$d < 0.8R$ or $K \geq 900$	$(d < 0.8R)$ or $(d < R$ and $i < R)$ or $(K \geq 100)$

The critical notion is whether the original rules leave any *room for maneuver*. This situation arises when the rules for a given object type do not apply in all possible situations described by the object type, corresponding to cases where human worker or manager would be called upon to make the actual loan decision. Inheritance means the addition of new properties and constraints; inheritance at the rules level means that parts of the rule's decision space that were left unspecified or in Permitted status may be in Obligatory or Forbidden status in the subclass.

Figure 4 show the decision space for a Private Loan (PL); since we are now dealing with 3 parameters (3 dimensions) we have to use two diagrams. Note that in the case for $i \geq r$, part of the "permitted" space actually becomes forbidden to grant, but there is still an open choice as these applications can be either forwarded or rejected.

3 Reasoning

To reason about business rules we represent them as constraints in the fashion described above: the condition simply implies a particular assignment. E.g., the constraint

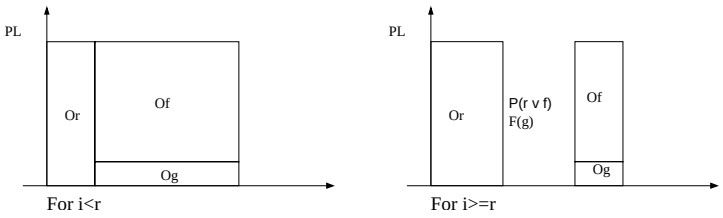


Figure 4. Private Loan

corresponding to the first rule in the table is $i < 0.5R \rightarrow Mandate = O \wedge Action = r$.

Apart from "firing" it as a rule in the usual fashion (following the implication), there are two other modes in which we can reason about the set of rules. Checking for consistency wih a given mandate and action after adding the incompatibility axiom mentioned above, i.e., $O^a_o \leftrightarrow \neg F^a_o$, and its super/subtype versions $O^a_{o'} \leftrightarrow \neg F^a_o$, $O^a_o \leftrightarrow \neg F^a_{o'}$, where o' is the subtype of o, is a check for consistency of the inheritance relationship.

Conversely, when using a constraint solver that can reason about algebraic epxressions, we could use it and the requirement that the subtype rule does not contradict the supertype rule to infer an appropriately restricted version of the subtype rule from a version that is not restricted enough. This can be used as an interactive tool to guide subtype rule development; the constraint solver guarantees that the subtype's rules will either correspond to an O or F mandate in the supertype in the same area of the decision space, or make a new O or F mandate fit in the "Permitted" or unspecified part of the supertype's rule's decision space.

References

[BGM00] Elisa Bertino, Giovanna Guerrini, and Isabella Merlo. Trigger inheritance and overriding in an active object database system. 12(4):588–608, July 2000.

[BS94] P. Bichler and M. Schrefl. Active Object-Oriented Database Design Using Active Object/Behavior Diagrams. In J. Widom and S. Chakravarthy, editors, *Proc. IEEE RIDE'94*, pages 163–171, Houston, 1994.

[CCS94] Christine Collet, Thierry Coupaye, and T. Svensen. NAOS - efficient and moduler reactive capabilities in an object-oriented database system. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 132–143, 1994.

[CMR98] Nauman A. Chaudhry, James R. Moyne, and Elke A. Rundensteiner. A formal model for rule inheritance and overriding in active object-oriented databases. In *Proceedings 3rd Integrated Design and Process Technology Conference (IADT '98)*, pages 128–135, 1998.

[KS95] Peter Kueng and Michael Schrefl. Spezialisierung von Geschaeftsprozessen am Beispiel der Beantragung von Kreditantraegen. *HMD*, 185, 1995.

[SS00] M. Stumptner and M. Schrefl. Behavior consistent inheritance in UML. In *Proc. Intl. Entity-Relationship Conference (ER 2000)*, Salt Lake City, October 2000.

[SS02] M. Schrefl and M. Stumptner. Behavior consistent specialization of object life cycles. *ACM Transactions on Software Engineering and Methodology*, 11(1):92–148, 2002.

[WC96] Jennifer Widom and Stefano Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann Publishers, Inc., 1996.

Interactive Configuration and Evaluation of a Heat Treatment Operation

M. Aldanondo[(a)], E. Vareilles[(a)] K. Hadj-Hamou [(b)] and Paul Gaborit [(a)]

[(a)] Centre de Génie Industriel - École des Mines d'Albi-Carmaux
Campus Jarlard - 81013 Albi CT Cedex 09 - France
[(b)] Laboratoire GILCO - ENSGI – INP Grenoble
46 Avenue Félix Viallet - 38031 Grenoble - France
{aldanondo, vareilles, gaborit}@enstimac.fr, hamou@gilco.inpg.fr

Abstract

This paper presents an interactive configuration system that allows solution evaluation. This work is driven by an industrial case dealing with heat treatment operation. The first part presents the problem and provides the main solution ideas. Then the configuration model mixing discrete and numerical constraints is presented. The third section provides filtering elements in order to permit interactive configuration. The last one discusses the designed configuration/evaluation system.

1. Introduction

The goal of this communication is to present an interactive configuration system that simultaneously allows interactive configuration and qualitative evaluation. This problem originates from a European project called VHT for "Virtual Heat Treatment" (project No G1RD-CT-2002-00835) whose purpose is to set up a Knowledge Based System able to configure and evaluate heat treatment operation.

1.1 – The industrial problem

A heat treatment operation consists in raising the temperature of a steel part until a certain temperature, remaining at this temperature for a while, then cooling down rapidly. The expected effect is an improvement of the mechanical properties of the part. But, simultaneously, a negative effect corresponding with part distortions occurs most of the time.

The means to evaluate distortions is computer-simulation thanks to Finite Element Method Codes (FEM). They allow obtaining quantitative prediction of the distortion at the design stage. However these tools are costly, time consuming and very complicated to use (due to mechanical/thermal/structural behaviours and data requirements). The situation is getting even more difficult in the current context of smaller production series and the high reactivity of the market. Due to the lack of satisfying simulation-based tools, prediction of distortion relies mostly on the know-how from heat treatment experts or users. Therefore the idea of collecting the know-how about heat treatment distortion and to organize it in a Knowledge Based System is at the origin of this work. The whole system gathers a CBR module and a configuration module. This communication deals with the configuration/evaluation module.

A heat treatment operation can be defined according to:

- part material, characterised by parameters relevant to chemical composition and mechanical/thermal/structural behaviours,
- part geometry, characterised by parameters relevant to the shape, massiveness, symmetry of the part,
- heat treatment conditions, characterised by parameters describing the heating device, the cooling system, the parts layout during heating and quenching and various tuning (duration, temperature, flow-rate…).

All these parameters are linked by constraints. Configuring a heat treatment operation corresponds with the selection of a value for each of these parameters with respect to all the constraints.

In order to have a configured heat treatment operation that satisfies the user, it is necessary to evaluate the improvement of the mechanical properties of the part but also to evaluate if the level of relevant distortion is acceptable. If the way to select the parameters defining the heat treatment operation in order to improve the mechanical properties is well known, distortion prediction is much more delicate. The scope of evaluation is therefore to qualitatively quantify distortion (i) during the configuration process in order to interactively assist the user and (ii) at the end of the configuration process in order to compare different configured heat treatment operations.

1.2 – The proposed solution

Most works related to configuration techniques concern physical products. The configuration of process operations has been much less studied. [Geneste *et al* 2000] proposed an application dealing with a machining operation. In their work, configuration is interactive and is used in an interactive process for aiding decision. In a similar way, the proposed application deals with the interactive configuration of a heat treatment operation. Furthermore, it computes, after each inputs, the intensity of the final distortion of the configuration problem, with some numerical constraints filtering techniques.

A way to compute the evaluation of a configuration solution is to assign weights to each parameter values and to each allowed combinations of parameter values relevant to constraints. Both the quality of the chosen values and the resulting associations of values usually influence the goodness of a solution. The evaluation of a solution is defined as the sum of the weights of all the values and pairs of values involved in the solution [Hulubei *et al* 2003]. The main interest of this approach lies in its simplicity and its optimisation possibilities. But in an interactive configuration process, this approach does not easily allow the user to input some constraints on the result and to see the consequences on the parameter values. In our case, the possibility to input a maximum value on distortion and

to configure with respect to this constraint is a necessity. Therefore, it was decided to embed the evaluation process in the configuration process. In our case, this means, that distortion and formulae that compute distortion are considered in the configuration model.

The resulting configuration model gathers:
- two sets of variables corresponding with:
  - heat treatment parameters {vp},
  - distortion attributes{vd},
- constraints linking:
  - only vp, necessary for heat treatment configuration,
  - vp and vd, necessary for computation of the distortion.

Two ways of using the configuration system are therefore possible. The first one consists in interactively inputting parameter values and getting the computation of the relevant distortion. The second one consists in first inputting parameter values that are not negotiable (part shape for example), then inputting some threshold on the maximum value of the distortion in order to get domain restrictions on negotiable parameters (heating rate for example).

1.3 – Paper organization

The second section presents the configuration model gathering two configuration and evaluation model pieces. It will show that three kinds of constraints are necessary: discrete constraints, numerical constraints and mixed constraints. As we target an interactive assistance, the filtering elements for each kind of constraints will be described in section three with the means to gather them in a single filtering engine. Then some interest and limits of the system will be discussed in the last section.

2 The knowledge model

The goal of this section is to present an overview of the configuration model. This section is divided in two parts. The first one describes the model piece necessary for configuring the heat treatment operation. The second one deals with the evaluation of distortion and shows how the two model pieces fit together.
For each sub-section, the model is described as a Constraint Satisfaction Problem (CSP) [Tsang 1993] and outlines the kind of variables and the kind of constraints that are necessary. Various models were designed with heat treatment experts, coming both from academia and industry, through a dozen of meetings planned during the past 18 months [David *et al* 2003]. The most advance one concerns parts that belong to a part family "axis" (one dimension much longer than the two others). Another model dealing with a part family "disc" (one dimension much smaller than the two others) is currently investigated.

2.1 – Piece of model relevant to heat treatment configuration

Around 50 parameters have been identified by the experts and characterize the material of the part, the geometry of the part or the heat treatment conditions. Each parameter is associated with a configuration variable upon which the user can input a domain restriction. These variables are either symbolic or numerical. Some typical variables are shown in the table 1.

	Symbolic
Part material	forming_process
	delivery_structure
	……
Part geometry	hole_existence
	shoulder_existence
	……
Heat treatment conditions	quenching_fluid_type
	gravity_direction
	……
	Numerical
Part material	carbon_percentage
	roughness_variation
	……
Part geometry	average_diameter
	thickness_variation
	……
Heat treatment conditions	quenching_fluid_temperature
	heating_rate
	……

Table 1 – Configuration variables

2.1.1 - Compatibility constraints

Most of the constraints relevant to heat treatment configuration are compatibility constraints expressing restrictions on variable values. As variables are either symbolic or numerical, different kinds of constraints are present.

Constraints between the symbolic variables correspond with compatibility tables as the one shown in table 2 providing the kind of fluid behaviour with respect to the quenching fluid type.

quenching_fluid_type	fluid behaviour
"water"	"turbulent"
"water"	"laminated"
"gas"	"turbulent"
….	…..

Table 2 – Discrete constraint

Constraints between numerical variables can correspond with :
(i) simple mathematical operations (+, - , * , /), as for example a temperature increase computation :
Delta_temp = Temp_max – Temp_min
(ii) compatibility tables, when heat treatment experts can provide compatibilities between intervals, as for example a compatibility between a furnace load and a heating speed shown in table 3.

furnace_load	heating_speed
[0, 500]	[15, 25]
[250, 750]	[10, 15]
[500, 1000]	[5, 10]
…..	….

Table 3 – Numerical constraint

or (iii) experimental curves or abacus, they require more complicated discretization techniques as Quad Trees proposed by [Sam 1995] that have been successfully investigated but will not be detailed in this communication.

Constraints mixing symbolic and numerical variables are always expressed thanks to compatibility table as the one in table 4

showing a relation between the fluid speed and the type of quenching fluid.

quenching_fluid_type	fluid_speed
"water"	[0.5, 2]
"oil"	[0.5, 2]
"gas"	[2, 10]
….	…..

Table 4 – Mixed constraint

2.1.2 - Activity constraints

In order to be able to modulate the existence of some parameters, it is necessary to be able to express some variable existence conditions. This kind of constraints, called activity constraint in the dynamic extension of CSP [Mittal and Falkenhainer 1990], have the following shape, X = "x" => Y exists, meaning if variable "X" equals the value "x" then variable "Y" exists. They have been mainly used to describe geometric parameters of the part as for example:
hole_existence = "yes" => Variable_hole_diameter exists.
part_thickness $\in$ [10 , 20] => Variable_part_area exists.

2.1.3 - Conclusions

For the model piece relevant to heat treatment configuration, we deal with (i) a set of variables vp that are symbolic or numerical, (ii) constraints that can be discrete, numerical or mixed and (iii) compatibility and activity constraints. Figure 1 (where circles are variables and lines correspond with constraints) illustrates this simple piece of model.

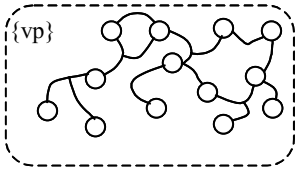


Figure 1 – Configuration model

2.2 – Piece of model relevant to the evaluation of distortion

As part distortion is geometrically very complicated to describe, heat treatment experts have proposed to quantify five basic distortion shapes: (i) "spool/barrel", (ii) "banana", (iii) "ovalization", (iv) "spacing/tightening" and (v) "umbrella", as shown in figure 2. It can be noted that (i) "spacing/tightening" distortion can exist, if the part has got a hole, in the left and/or right part extremities and (ii) "umbrella" distortion can exist if shoulder shapes are present.

2.2.1 – Evaluation variables

Each of these five distortion components is associated with some kind of grade. In order to calculate such a value, heat treatment experts propose the following approach based on three kinds of distortion attributes:
- a first small subset of heat treatment parameters, vp, allows a first quantification of each distortion component, named potential distortion attribute, vd_pot,
- a second larger subset of heat treatment parameters, vp, permits to quantify intermediate distortion attributes, vd_int. These attributes

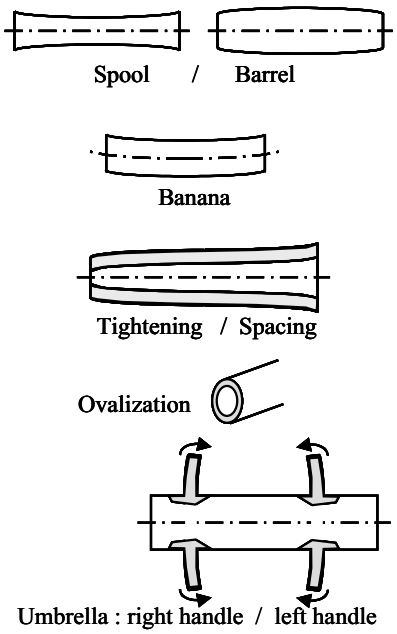


Figure 2 – Five distortion types

modulate each potential distortion component in a similar way. Around 26 intermediate distortion attributes have been identified.
- the resulting grade of each distortion component, named final distortion attribute vd_fin, is computed as a product of the relevant potential distortion attribute and the set of intermediate distortion attributes.
This is illustrated in figure 3 with a model gathering two distortion components (two couples (vd_pot , vd_fin)) and three intermediate distortion attributes (vd_int). The solid lines represent the possible relation between parameters and various distortion attributes.

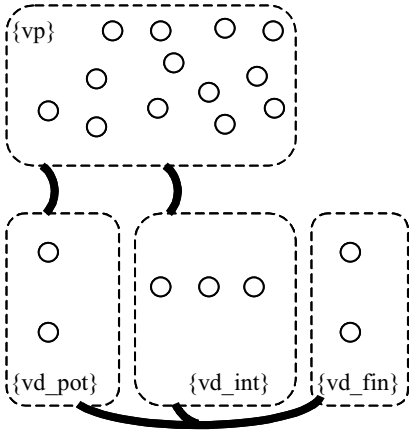


Figure 3 – Three kinds of distortion attributes

Each distortion attribute is associated with a configuration variable. These variables are all numerical and defined within intervals. In order to compare solutions, heat treatment experts decided that (i) each final distortion component (vd_fin) is quantified into an interval [1, 1000], where "1" means no distortion and "1000" means maximum distortion, (ii) each potential distortion component (vd_pot) is quantified into an interval [1, 20] while (iii) each intermediate distortion attribute (vd_int_k) is quantified into intervals [1, α_k] such their product belongs to an interval [1, 50].

2.2.2 – Evaluation constraints

As potential and intermediate distortion attributes are all numerical and parameters either symbolic or numerical, constraints linking these distortion attributes with parameters are either mixed or numerical. All these constraints have been expressed thanks to compatibility tables: (i) 5 constraints for the quantification of the 5 components of potential distortion (vp, vd_pot) and (ii) 26 constraints for the quantification of the 26 intermediate distortion attributes (vp, vd_int).

As each final distortion component is obtained with the computation of the product of the relevant potential distortion component and the set of intermediate distortion attributes, the following numerical constraint: $vd\_fin = vd\_pot * \Pi_{k=1 \text{ to } 26}(vd\_int_k)$ must be defined for each of the five distortion components.

This is illustrated in the lower part of figure 4 with the example of model of figure 3.

2.3 – Gathering the two model pieces in a single model

The two model pieces are gathered in a single configuration/evaluation model (figure 4) showing:

- two kinds of variable associated with:
  - parameters (vp): around 70 symbolic and numerical variables,
  - distortion attributes (vd_pot, vd_int, vd_fin): around 40 numerical variables,
- three kinds of constraints between:
  - parameters (vp): around 40 constraints, gathering activity and compatibility constraints, which can be symbolic, numerical or mixed constraints and mainly expressed thanks to compatibility tables,
  - parameter (vp) and distortion attributes (vd_pot, vd_int): around 50 compatibility constraints which can be numerical or mixed constraints and which are described as compatibility tables,
  - distortion attributes (vd_pot, vd_int, vd_fin), around 60 constraints which are numerical compatibility constraints,

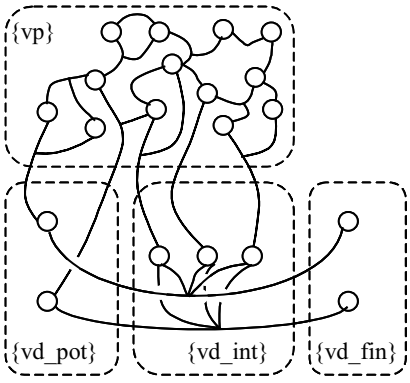


Figure 4 – Global configuration/evaluation model

and defined with the "product" mathematical operation,

The object of the next section is to study how this model can be used to interactively and simultaneously allows configuration and evaluation of the heat treatment operation.

3 - Filtering techniques

The object of this section is to present the various filtering techniques that have been necessary and show how they have been gathered in a single filtering engine.

3.1 - Discrete constraints defined with compatibility tables

A simple arc consistency technique (AC-3) has been used to propagate this kind of constraints. As some constraints have an arity larger than two, AC-3 has been adapted. The filtering techniques work as follows:

1-Input on variable var-x
2- Put var-x in list-var-1
3- While list-var-1 non empty:
    4- Take variable var-i of list-var-1
    5- Put constraint including var-i in list-cst-1
    6- While list-cst-1 non empty:
        7- Take cst-j of list-cst-1
        8- Filter cst-j with var-i
        9- Put var, with a reduced domain, in list-var-1

3.2 - Mixed constraints defined with compatibility tables

For compatibility tables including numerical variables defined with intervals, the association of a label with each interval permits to consider a discrete constraint [Faltings 1994]. The previous filtering means (3.1) can be used except that when the definition domain of a numerical variable is reduced, it is necessary to reconstruct the definition domain of the variable with classical set operations (union, intersection). Filtering therefore goes as follows :

        7- Take cst-j of list-cst-1
        8- Filter cst-j with var-i
        8'- Reconstruct definition domain of the reduced var
        9- Put var, with a reduced domain, in list-var-1

3.3 – Continuous constraints defined with compatibility tables

The previous labelling and domain reconstruction means are used in order to filter this kind of constraints.

3.4 - Continuous constraints defined with simple formulae

When all the variables are numerical and each constraint written with a mathematical formula, $f(x_1, x_2, \ldots x_n) = 0$, B-Consistency, proposed by [Lhomme 1993] and based on interval analysis [Moore 1966], proposes filtering techniques that operate fine if:
(i) $f(x_1, x_2, \ldots x_n) = 0$ can be projected on any variable x_i meaning that a function f_i exists as: $x_i = f_i(x_1, x_2 \ldots . x_{i-1}, x_{i+1}, \ldots x_n)$
(ii) any projection f_i is continuous and monotonous.
(iii) only one constraint expressed as a formula acts on a same variable subset. B-Consistency is weak when more than one constraint acts on a same sub-set of variables (corresponding to some constraint intersection). It is shown in [Lhomme and Rueher 1997] that a simple problem gathering two variables and three constraints cannot be fully filtered.
(iv) each variable occurs only on time in a formula. In the opposite case, for example: $x1^2 - x1 - x2 = 0$, it has been shown that the way to express the mathematical expression, for example: $x1^2 - x1 - x2 = 0$ or $x1*(x1 - 1) - x2 = 0$, influences the quality of the filtering operation [Lhomme, Rueher, 1997].

As this kind of constraints is only present in our model: (i) between configuration parameters, vp, with simple mathematical operation (+, - , * , /) and (ii) for computing the product quantifying the final distortion attribute (vd_fin = vd_pot * $\Pi_{1 \text{ to } 26}$(vd_int)), the previous restrictions are not present and B-Consistency can be used. For a single continuous constraint, filtering is done as follows:

1- Input on variable var-x
2- Put var-x in list-var-1
3- While list-var-1 non empty
    4- Take variable var-i of list-var-1
    5- Put all constraint projections including var-i in list-cst-1 except the one relevant to var-i
    6- While list-cst-1 non empty:
        7- Take constraint projections of list-cst-1
        8- Compute the resulting domain of the projected variable
        8"- Intersect resulting domain with initial domain of the projected variable
        9-If domain is reduced, put projected variable in list-var-1

3.5 – Activity constraints

In order to deal with activity constraints, we only need to use the "Require Variable" activity constraint (var-x = "x" => var-y exists) among the four categories (Require, Require not, Always require, Always require not) proposed by [Mittal and Falkenhainer 1990] Resulting filtering goes as follows:

1- Input on variable var-x
2- Put var-x in list-var-1
3- While list-var-1 non empty:
    4- Take variable var-i of list-var-1
    5- Put activity constraint including var-i in its premise (left part of =>) in list-a-cst-1
    6- While list-a-cst-1 non empty:
        7- Take a-cst-j of list-a-cst-1
        8- Evaluate the premise of a-cst-j
        9- If true, add triggered var-y (right part of =>) in the current problem and put this variable in list-var-1

3.6 – Gathering the filtering techniques in a single engine

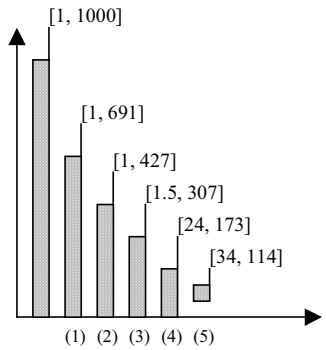
The three filtering techniques have been gathered in a single engine. It was decided to first filter activity constraints, then constraint expressed with compatibility tables to finish by constraints defined by formulae. The architecture of the global filtering is therefore as follows:

Input on var
var in list-var-1
While list-var-1 non empty
    Take variable var-i of list-var-1
    Filter activity constraint (section 3.5)
        Result: can add variables to the current problem and put them in list-var-1
    Filter discrete, mixed, continuous constraints defined with compatibility tables (section 3.1, 3.2 and 3.3)
        Result: put variables with a reduced domain in list-var-1
    Filter Continuous constraints defined with formulae (section 3.4)
        Result: put variables with a reduced domain in list-var-1

4 – Interest and limits of the proposed approach

All these elements have been set up in a software mock-up that can be seen on the web at: http://iena.enstimac.fr:20000/cgi-bin/vht.pl. Two ways to use the configuration system have been identified at the end of section 1.2. The first one, consisting in interactively inputting restrictions only on parameter and see the computation of the relevant distortion raises an interesting point. While the other, inputting parameter restrictions (that are not negotiable) and some threshold on the maximum value of the distortion in order to get domain restrictions (on negotiable parameters), points out a limit of the approach.

During the interactive configuration process, each time the user inputs a restriction on a variable corresponding with a parameter (vp), constraints are filtered and other parameters (vp) and/or distortion attributes (vd) have their domain reduced without any problem. If this behaviour is normal and expected in interactive configuration for parameters (vp), it is of interest for distortion evaluation (vd).
At the beginning of configuration process, each final distortion attribute has a definition domain equal to the interval [1, 1000] meaning that no information about distortion level is available. As filtering is launched after each user input, each final distortion domain is progressively reduced during the configuration process. The example of figure 5 shows how the intensity of the banana distortion decreases when, first, the geometry of the part is described (1), then, the quenching fluid.(2), the load preparation (3),



the material (4) and finally, the metallurgical characteristics (5).

Figure 5 – Example of the evaluation of banana intensity

Even if sometimes this definition interval is split in different intervals, this allows the user to see the progressive effects of his selections on each final distortion component. This interesting behaviour is allowed by B-Consistency that operates with interval analysis. At the end of the configuration process, each final distortion attribute has a domain corresponding with a reduced interval, as shown by the row (5) in figure 5. An average between the lower and upper interval bounds can be calculated for solution comparison.

The previous good point has a drawback. If the user inputs restrictions that affect variables corresponding with parameters (vp) and final distortion attributes (vd_fin), the proposed filtering approach can lead to an inconsistent problem.
Let us consider the following problem defined with:

- 3 parameters with relevant definition domain, vp_1 with D(vp_1) = {“yes”, “no”}, vp_2 with D(vp_2) = {“1”, “2”} and vp_3 with D(vp_3) = {“A”, “B”}
- 3 distortion attributes with relevant definition domain, vd_pot with D(vd_pot) = [1, 4], vd_int with D(vd_int) = [1, 3] and vd_fin with D(vd_fin) = [1, 12],
- 1 discrete constraint between parameters: (vp_2, vp_3) = {(“1”, ”A”), (“2”, “B”)},
- 2 mixed constraints between parameters and distortion attributes: (vp_1, vp_2, vd_pot) = {(“yes”, “1”, [3, 4]), (“no”, “1”, [2, 3]), (“no”, “2”, [1, 2])} and (vp_3, vd_int) = {(“A”, [1,2]), (“B”, [2,3]) ,
- 1 numerical constraint between distortion attributes:

vd_pot * vd_int = vd_fin

and represented in figure 6.

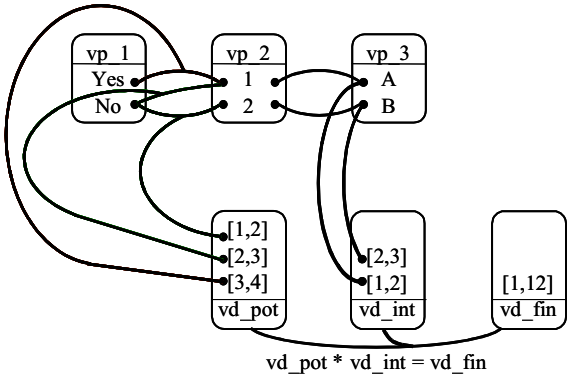


Figure 6 – Example leading to an inconsistency

If the user inputs the restriction vd_fin < 2 (or vd_fin = [1, 2]), the filtering of the numerical constraint provides:
- vd_pot = vd_fin/vd_int = [1, 2]/[1,3] ∩ [1,4] = [1,2]
- vd_int = vd_fin/vd_pot = [1, 2]/[1,2] ∩ [1,4] = [1,2]
- vd_pot = [1,2] => vp_1 = {“no”} and vp_2 = {“2”}
- vd_int = [1,2] => vp_3 = {“A”}

This problem is inconsistent because vp_2 = {“2”} and vp_3 = {“A”} does not respect the constraint between the two parameters (vp_2, vp_3). A same result would be got with the restriction vd_fin > 8. This problem comes from the arc consistence filtering techniques that consider sequentially only one constraint at a time and only check the consistence of pair of variables. Filtering with a stronger consistency could avoid this problem but would be too much time consuming.

5 – Conclusions

The goal of this communication was to present an interactive configuration system that simultaneously allows interactive configuration and qualitative evaluation of heat treatment operations.

A knowledge model gathering a configuration model piece and an evaluation model piece was designed. The corresponding constraint satisfaction problem uses discrete and numerical variables, discrete, mixed and numerical constraints. Classical filtering techniques, for discrete constraints (compatibility tables), have been embedded with B-consistence, for numerical constraints (formulae), in a single filtering engine.

The resulting configuration software allows the user to input restrictions either on the parameters defining the product to configure or on the attributes that characterize the evaluation of the configured product.

The model and configuration software are now in a validation step performed by end users. In spite of the aiding decision tool, users underline the fact that they better understand heat treatment process. The drawback, discussed in section 4, is avoided during configuration by progressively reducing final distortion attributes with a “try and error” procedure.

In terms of knowledge modelling, it can be pointed out that heat treatment experts succeeded rather quickly to deal with the formalism variable/domain/constraint. This confirms, as frequently reported in other papers, that the natural and easy to understand concept of the CSP approach makes it a good candidate for domain knowledge representation.

For this project, the works remaining to be done deal with knowledge model validation and integration of the configuration system with the case base reasoning system.

References

[David *et al* 2003] Ph. David, M. Veaux, E. Vareilles and J. Maury - Virtual Heat Treatment Tool for Monitoring and Optimising Heat Treatment Process - 2nd International Conference on Thermal Process Modelling and Computer Simulation, Nancy, 2003.

[Faltings 1994] B.Faltings - Arc consistency for continuous variables, dans Artificial Intelligence, volume 65, pages 363-376, 1994.

[Geneste *et al* 2000] L. Geneste, M. Ruet and T. Monteiro - Configuration of a machining operation - ECAI Workshop on Configuration, Berlin, Germany, pp. 44-49, 2000.

[Hulubei *et al* 2003].T. Hulubei, E.C. Freuder and R.J. Wallace.- The Goldilocks problem - AI EDAM, Volume 17, Number 1, January 2003 pp 3-11

[Lhomme 1993] O. Lhomme - Consistency techniques for numerical CSPs - IJCAI 93, Chambéry, France, 1993.

[Lhomme and Rueher 1997] O. Lhomme and M.Rueher - Application des techniques CSP au raisonnement sur les intervalles - Revue d’intelligence artificielle, Dunod, Vol. 11, pp. 283-311, 1997.

[Mittal and Falkenhainer 1990] S. Mittal and B. Falkenhainer - Dynamic Constraint Satisfaction Problems - 9th National Conference on Artificial Intelligence AAAI, Boston, USA, 1990, pp. 25-32.

[Moore 1966] R.E Moore - Intervals Analysis - Prentice Hall, 1966.

[Sam 1995] D. Sam - Constraint Consistency Techniques for Continuous Domains - PhD Thesis EPFL, Lausanne, Switzerland, 1995.

[Tsang 1993] E. Tsang - Foundations of constraints satisfaction - Academic Press, London, 1993.

STAR-IT: a Tool to Build STAR Applications

Diego Magro
Dipartimento di Informatica, Università di Torino
Corso Svizzera 185; 10149 Torino; Italy
magro@di.unito.it

Abstract

Intelligent Web-based systems exploiting advanced problem solving techniques to support the users in complex tasks have to face three main challenges: (i) Non-expert users should not be exposed to the complexity of the knowledge representation and of the reasoning mechanisms used by the problem solver. (ii) The system should not be committed to any specific approach. (iii) A tool supporting the domain expert in the instantiation of the system on new domains and in system maintenance is needed. We claim that this three goals require both a careful conceptualization of the application domain and an intermediate representation layer. This layer has the role of filling the gap between the implementation-oriented view of the domain, needed by the reasoning module, and the human-oriented view of the same domain, needed by the frontend of both the instantiation tool and the final system. This intermediate layer also provides a representation which is independent from the specific reasoning approach adopted. In the paper we discuss this proposal by referring to a specific example, i.e. the STAR system (a Web-based system which exploits a configuration engine to support the user in organizing a personalized tour) together with STAR-IT, a tool that supports the instantiation of STAR on different domains.

1 Introduction

People surfing the Web require an active and intelligent support, even in complex tasks (e.g., the Web should provide financial or medical advice, leasure and entertainment offers, travel plans and so on). For this reason, Web-based systems need to be equipped with complex reasoning mechanisms and problem solving capabilities.

To avoid ad hoc solutions, the exploitation of existing problem solvers is recommended, also to ensure a certain degree of generality and flexibility which guarantees the possibility to configure (in the following we will use the term *instantiate*) the system on different domains.

Many modern systems exploiting problem solving techniques as well as the tools provided to build these systems (e.g., configuration systems and tools described in [Mailharro, 1998; Fleischanderl *et al.*, 1998; Junker and Mailharro, 2003; Bergenti, 2004]) are typically designed for expert users, who know the ontology behind the problem solver. Although this can be adequate in many applications, when the problem solving capabilities are embedded into Web systems, the interaction needs both of the final users and of the domain experts that provide the contents of these systems cannot be ignored.

This scenario poses three main challenges, to be faced when designing and developing an intelligent Web-based system that supports the user in a complex problem solving task (especially in non-technical applications):

- Non-expert users should not be exposed to the complexity of the knowledge representation and of the reasoning mechanisms used by the problem solver. In fact, both the knowledge representation and the reasoning mechanisms could be quite complex, expecially if the system exploits an existing problem solver (a configurator, a planner, or whatever), which was designed to be general with respect to the specific application domain.
- The system should not be committed to any specific approach, i.e., it should be designed in order to work with different problem solvers of the same type (e.g. different planners, possibly representing the domain knowledge in different ways), as well as with different types of problem solvers, since a same complex task may have more than one sound formalizations (e.g., some tasks may be formalized both as planning problems or as a scheduling problems). In other words, the reasoning engine should be wrapped within an independent module, in order to support the interoperability of the system with other modules, implementing different reasoning approaches.
- Another important aspect concerns the instantiation of the Web system on a particular domain and its maintenance. Such tasks are usually performed by a domain expert who is not aware of the reasoning techniques nor of the particular knowledge representation used by the problem solver. Thus, a tool supporting the domain expert in these tasks is needed.

We claim that this three goals require both a careful conceptualization of the application domain and an intermediate

knowledge representation layer. This layer has the role of filling the gap between the implementation-oriented view of the domain, needed by the reasoning module, and the human-oriented view of the same domain, needed by the frontend of both the instantiation tool and the final system. This intermediate layer also provides a representation which is independent from the specific reasoning approach adopted. In other words, such a layer will guarantee both the tranparency of the reasoning mechanisms for the user (domain expert and final user) and the independency of the frontend of the system (intantiation tool and final system) from the specific reasoning techniques adopted.

The design of such a layer requires a domain ontology, based on the concepts used by people to reason about that domain, coupled with a translation module, that provides the mapping between these concepts and the representation needed by the actual reasoning engine.

Similar issues have been discussed in other works, relevant to Web-based configuration systems (see, for instance, the CAWICOMS Workbench [Ardissono *et al.*, 2003]).

In the following discussion we refer to a specific example, i.e. STAR (Smart Tourist Agenda Recommender), a Web-based system which exploits a configuration engine to support the user in organizing a personalized tour in a given geographical area. In particular, we will discuss the main functionality of STAR-IT, a tool that supports the instantiation of STAR on different domains.

In Section 2 we introduce STAR, while in Section 3 we describe the istantiation tool; Section 4 conludes the paper.

2 STAR

In the travel and tourist domain different systems have been developed to support the users to plan long trips, involving flights scheduling and hotel reservation, or even to select a set of tourist activities (e.g. [Torrens *et al.*, 2002; C.Knoblock, 2004; Ricci *et al.*, 2002]). However, in such systems the task of organizing a coherent agenda is totally left to the user.

The organization of a tourist agenda is a task that people are quite used to deal with. However, accomplishing such a task in a satisfactory way can be complex and time consuming, since it requires to access different information sources (tourist guides, the Web, local newspapers, etc.), to consider various kinds of constraints (the available time, the physical locations in which the different activities take place, and so on), and to mentally backtrack many times (e.g. when discovering that an interesting museum is closed, that two historical buildings are too far from each other to be visited in the same morning). Within our prototype STAR [Goy and Magro, 2004], we have tried to automate the task of organizing a tourist agenda by defining it as a configuration problem [Sabin and Weigel, 1998] and exploiting a configuration engine based on the $\mathcal{FPC}$ framework [Magro and Torasso, 2003], to find a solution that fulfils user requirements.

STAR's reasoning mechanisms are based on a declarative representation of a generic agenda as a complex (i.e. configurable) entity in which tourist items (e.g. buildings, restaurants, etc.) are the basic components and denote the corresponding activities (e.g. visits to buildings, lunch in a restaurant, etc.). The constraints on the components are declaratively expressed in STAR's knowledge base by means of $\mathcal{FPC}$ constraint language. Similarly, each user requirement (stated by the final user) is coded as a set of $\mathcal{FPC}$ (input) constraints. STAR's reasoning engine takes as input both the knowledge base and the input constraints (representing the user requirements) in order to configure a suitable agenda to suggest to the tourist (i.e. the final user of the system).

3 STAR-IT

STAR-IT is a tool supporting the domain expert in instantiating STAR on particular domains. Two STAR applications may differ w.r.t. various dimensions, which represent the degrees of freedom of the instantiation activity. Here we focus on those dimensions that are most closely related to the problem solving capability: the description of the general structure of an agenda and the definition of the properties that the final user can impose as requirements.

We will show how the domain expert can instantiate a system like STAR without being a knowledge engineer, and in a way that is totally independent from the specific inference engine actually exploited by the system. The instantiation process is depicted in Figure 1.

Thanks to STAR-IT, the domain expert can describe the general structure of an agenda and its possible content by means of common concepts, which include: the time slots of the day (e.g. morning, afternoon, etc.); the tourist items (e.g. buildings, restaurants, events, etc.) that are involved in the tourist activities (e.g. visiting buildings, eating, attending events, etc.); the possible durations of the activities; the daily opening hours and the actual tourist attractions availability in particular days; the distance between the physical locations in which the activities take place.

STAR-IT produces an intermediate representation based on these concepts, which expresses the user-oriented view of the domain and is independent from the specific reasoner. A module, called Adapter, plays the role of a bridge between the problem solver and the rest of the system. Such module takes as input the intermediate representation and it produces a knowledge base in the problem solver format (therefore the Adapter depends on the particular problem solver). In the current prototipe, a $\mathcal{FPC}$ general configurator plays the role of the problem solver and the Adapter translates the intermediate description of the general agenda into a $\mathcal{FPC}$ knowledge base. Such a knowledge base is expressed by means of the usual configuration concepts: configurable and basic components and subcomponents, partonomic relations, constraints among (sub)components and so on. The domain expert can be unaware both of the configuration ontology and of the ways the configuration concepts are actually represented within the configurator.

In the following we show a fragment of the internal representation, based on an XML language, adopted by STAR-IT at the intermediate layer:

```
<DAILY_AGENDA>
  <TIME_SLOTS>
    <TIME_SLOT>
      <TS_NAME>Morning</TS_NAME>
```

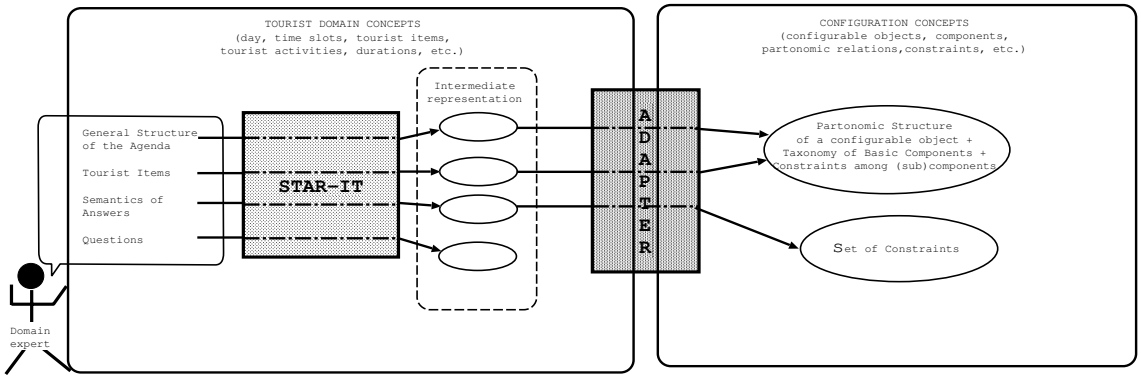


Figure 1: Instantiation Process

```
  <TS_MAX_DURATION>
    12
  </TS_MAX_DURATION>
  <TS_PREFERRED_DURATION>
    10
  </TS_PREFERRED_DURATION>
  <ACTIVITIES>
    <ACTIVITY>
      <A_NAME>havingBreakfast</A_NAME>
      <A_MIN_NUM>0<A_MIN_NUM>
      <A_MAX_NUM>1</A_MAX_NUM>
      <A_ITEM_TYPE>Bar</A_ITEM_TYPE>
      ...
    </ACTIVITY>
    <ACTIVITY>
      <A_NAME>visitingAttractions</A_NAME>
      <A_MIN_NUM>0<A_MIN_NUM>
      <A_MAX_NUM>10</A_MAX_NUM>
      <A_ITEM_TYPE>
        TouristAttraction
      </A_ITEM_TYPE>
      ...
    </ACTIVITY>
    <ACTIVITY>
      <A_NAME>havingLunch</A_NAME>
      ...
    </ACTIVITY>
    <ACTIVITY>
      <A_NAME>doingShopping</A_NAME>
      ...
    </ACTIVITY>
    <ACTIVITY>
      <A_NAME>attendingEvent</A_NAME>
      ...
    </ACTIVITY>
  </ACTIVITIES>
</TIME_SLOT>
<TIME_SLOT>
  <TS_NAME>Afternoon</TS_NAME>
  ...
</TIME_SLOT>
<TIME_SLOT>
  <TS_NAME>Evening</TS_NAME>
  ...
</TIME_SLOT>
  </TIME_SLOTS>
  ...
</DAILY_AGENDA>
```

The part enclosed in the DAILY_AGENDA tags describes the general structure of a one-day agenda. This fragment, for instance, says that a day is partitioned in three time slots (Morning, Afternoon and Evening). The morning should better not exceed 10 time units (the domain expert can define her own time unit; in this example, each time unit roughly corresponds to half an hour) and it must not be longer that 12 time units. The set of all the possible activities is specified for each time slot of the day; e.g. in the morning a tourist can have breakfast, visit up to 10 tourist attractions and so on. The tourist items that can be involved in the activities are described in the section enclosed between the AVAILABLE_TOURIST_ITEMS tags and they are organized into a taxonomy, as shown in the following:

```
<AVAILABLE_TOURIST_ITEMS>
  <CLASSES>
    <CLASS>
      <C_NAME>Bar</C_NAME>
        <SUBCLASS_OF>FoodPlace</SUBCLASS_OF>
      ...
    </CLASS>
    ...
  </CLASSES>
  <ELEMENTS>
    <ELEMENT>
      <E_NAME>HappyBar</E_NAME>
      <IS_A>Bar</IS_A>
    </ELEMENT>
  <ELEMENTS>
<AVAILABLE_TOURIST_ITEMS>
```

The Adapter translates this user-oriented representation into the representation required by the configurator. In particular, it builds a $\mathcal{FPC}$ partonomy representing the configurable daily agenda which, in this example, has three configurable direct components corresponding to the morning, the afternoon and the evening. These components have a set of partonomic roles representing the allowed activities and that can be filled with the tourist items, which are the basic components of the configuration. Moreover, a set of

$\mathcal{FPC}$ constraints restrict the set of valid configurations; e.g. the maximum and the preferred duration of the morning is coded by two weighted constraints (one with $k = 10$ and $w \neq \infty$ and the other with $k = 12$ and $w = \infty$), as follows:

$$\Sigma(\{\langle havingBreakfast, duration\rangle, \langle visitingAttractions, duration\rangle, \langle havingLunch, duration\rangle, \langle doingShopping, duration\rangle, \langle attendingEvent, duration\rangle\}) \leq k, weight = w$$

STAR-IT supports the domain expert also in defining the possible (final) user's requirements on a tourist agenda, i.e. which properties of the agenda are influenced by user's preferences and needs (e.g. having/not having breakfast, visiting historical museums, etc.), expressed in an initial interview (Web form) that STAR proposes to the user.

STAR-IT enables the domain expert to list the set of questions that STAR will ask to the final user, along with a finite set of possible answers. Each answer represents a user requirement whose meaning is specified during the instantiation phase.

To express the semantics of each answer, the domain expert has to identify which time slots, which activities, and which kinds of tourist items the requirement refers to; moreover, she has to specify the actual content of the requirement. For instance, in the following we show the intermediate representation of a requirement stating that the tourist is interested to visit many baroque buildings either in the morning or in the afternoon.

```
<REQUIREMENT>
  <TIME_SLOTS>
    <TIME_SLOT>Morning</TIME_SLOT>
    <TIME_SLOT>Afternoon</TIME_SLOT>
  </TIME_SLOTS>
  <ACTIVITIES>
    <ACTIVITY>visitingAttractions</ACTIVITY>
  </ACTIVITIES>
  <TOURIST_ITEMS>
    <ITEM_TYPE>
      <CLASS>Building</CLASS>
      <RESTRICTIONS>
        <RESTRICTION>
          <ATTRIBUTE>Style</ATTRIBUTE>
          <VALUE>Baroque</VALUE>
        <RESTRICTION>
      </RESTRICTIONS>
    </ITEM_TYPE>
  </TOURIST_ITEMS>
  <CONTENT>many</CONTENT>
</REQUIREMENT>
```

In the current prototype, the intermediate representation of each requirement is translated into a set of $\mathcal{FPC}$ constraints.

In Figure 1, an intermediate representation is depicted for the list of questions too. We do not describe it here, since it is used to produce the initial Web form the end user should fill in and it is not directly related to the configuration engine.

4 Discussion and Conclusions

In this paper we have presented STAR-IT, a tool supporting the instantiation of STAR (a Web-based system supporting the user in organizing a personalized tourist agenda) on new domains, and we have discussed how STAR-IT faces two challenges: (i) it hides the technological complexity to the user (i.e. the domain expert), who need not to be aware of the actual inference mechanisms underlying the system; (ii) it is not committed to a specific reasoning module, therefore modules based on different inference techniques can be easily plugged in the system. In particular, we have shown how an intermediate representation layer, representing the human-oriented view of the domain, can be exploited to achieve these goals.

STAR-IT and STAR applications share some objectives with the CAWICOMS Workbench for the development of distributed Web-based configuration services [Ardissono *et al.*, 2003]. The CAWICOMS framework concerns the distributed configuration of complex products and services, such as telecommunication switching systems and IP-based Virtual Private Networks. The harmonization of the user's and the system's points of views plays an important role also in the CAWICOMS approach, which suggests an adaptive user interaction mechanism in order to fill this gap. Moreover, a common configuration ontology is used to allow the distributed configurators to share pieces of information. However, besides the fact that STAR applications make use of a centralized problem solver, some other differences hold between them and the CAWICOMS systems. While the configuration capability is an essential aspect of the latter systems, STAR does not commit to the formalization of its task as a configuration problem (even if this is quite a natural way of formalizing it). An intermediate knowledge representation layer based on a general configuration ontology would make STAR independent of the particular configurators, but by itself it would not allow other kinds of problem solvers (e.g. planners or schedulers) to be easily plugged in. Therefore, an intermediate knowledge representation layer, independent of the specific formalization of the task, is needed. Moreover, the domain expert that defines the knowledge base is usually not aware of the technology behind the problem solver. Therefore, to be general w.r.t. the different valid task formalizations, while providing a simple way of representing the knowledge inserted by the domain expert, the above-mentioned STAR's intermediate knowledge representation layer is based on the concepts commonly used by people to describe tourist agendas.

The mechanisms for eliciting the users' requirements and preferences and to present the solution in a suitable way are central in the CAWICOMS framework. Instead, as we have said in the paper, almost all people are familiar with the task of drafting a tourist agenda, thus STAR applications do not require complex machinery to ease the elicitation of the users' requirements and the presentation of the solution (i.e. the suggested agenda), since both the requirements and the solutions can be expressed by means of the common concepts used in the intermediate knowledge representation layer.

STAR-IT is currently under development: as soon as the implementation is complete, we will perform an evaluation with domain experts, aimed at testing the impact of our architecture on the usability of the tool.

References

[Ardissono *et al.*, 2003] L. Ardissono, A. Goy, G. Petrone, A. Felfernig, G. Friedrich, D. Jannach, M. Zanker, and R. Schäfer. A framework for the development of personalized, distributed web-based configuration systems. *AI Magazine*, 24(3):93–110, 2003.

[Bergenti, 2004] F. Bergenti. Product and servive configuration for the masses. In *Proc. ECAI 2004 Configuration WS*, pages 7/1–7/6, 2004.

[C.Knoblock, 2004] C.Knoblock. Building software agents for planning, monitoring, and optimizing travel. In *Proc. of ENTER 2004*, 2004.

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

[Goy and Magro, 2004] A. Goy and D. Magro. Star: a smart tourist agenda recommender. In *Proc. ECAI 2004 Configuration WS*, pages 8/1–8/7, 2004.

[Junker and Mailharro, 2003] U. Junker and D. Mailharro. The logic of ilog (j)configurator: Combining constraint programming with a description logic. In *Proc. IJCAI-03 Configuration WS*, pages 13–20, 2003.

[Magro and Torasso, 2003] D. Magro and P. Torasso. Decomposition strategies for configuration problems. *AIEDAM, Special Issue on Configuration*, 17(1):51–73, 2003.

[Mailharro, 1998] D. Mailharro. A classification and constraint-based framework for configuration. *AI EDAM*, 12(4):383–397, 1998.

[Ricci *et al.*, 2002] F. Ricci, B. Arslan, N. Mirzadeh, and A. Venturini. Itr: a case-based travel advisory system. In *Proc. of ECCBR 2002*, pages 613–627, 2002.

[Sabin and Weigel, 1998] D. Sabin and R. Weigel. Product configuration frameworks - a survey. *IEEE Intelligent Systems*, (July/August 1998):42–49, 1998.

[Torrens *et al.*, 2002] M. Torrens, B. Faltings, and P. Pu. *SmartClients*: Constraint satisfaction as a paradigm for scaleable intelligent information systems. *Int. Journal of Constraints*, 7(1):49–69, 2002.

Kumbang Configurator–A Configuration Tool for Software Product Families*

Varvana Myllärniemi and **Timo Asikainen** and **Tomi Männistö** and **Timo Soininen**
Helsinki University of Technology
Software Business and Engineering Institute (SoberIT)
varvana.myllarniemi@hut.fi

Abstract

This paper presents Kumbang Configurator, a prototype system for deriving product individuals from configurable software product families. Configurable software product families resemble configurable products in that they have a pre-defined structure and can be customised according to customer requirements in a routine manner. The conceptual basis underlying the configurator is Kumbang, a language for modelling configurable software product families from the feature and architectural points of view. Features represent the family from a requirements or functional point of view, and architecture from a technical or structural one. The configurator has been implemented in the Java programming language, and validated with two examples, one of which is based on an industrial case.

1 Introduction

Software product families, or *software product lines*, as they are also called, have emerged as an important paradigm for software reuse [Clements and Northrop, 2001; Bosch, 2000]. A software product family is commonly defined to consist of a *common architecture*, a *set of reusable assets* used in systematically producing individual products, and the *set of products thus produced* [Bosch, 2000].

The most systematic class of software product families can be called *configurable software product families* [Bosch, 2002; Raatikainen *et al.*, 2005]. A configurable software product family has the property that individuals of the family can be deployed in a systematic manner; there is no need for coding within components, and not much need for adding glue code between them; hence, configurable software product families resemble configurable, non-software products. The deployment of individuals is called the *configuration task*. It has been noted that the configuration of software product families task can be burdensome and error-prone [Clements and Northrop, 2001]. Thus there is a need for concrete tool support.

*The support of National Technology Agency of Finland is acknowledged.

Product configurators, or *configurators* for short, are tools that support the configuration task in producing a correct product individual. Configurators prevent configuration errors and automate routine tasks. A large number of configurators have been developed, e.g. [Tiihonen *et al.*, 2003], mainly for the purpose of configuring non-software products.

The question whether configurators developed for non-software configurable products can be applied to software product families has been studied in [Asikainen *et al.*, 2003; 2004]. The outcome from these studies has been that the existing tools cover important parts of the configuration models of software product families, but lack support for architectural connections. Hence, new tools are needed.

In this paper, we introduce Kumbang Configurator, a tool supporting the configuration task for configurable software product families. Kumbang Configurator utilises Kumbang modelling language. Kumbang is based on modelling a configurable software product family from two independent, yet mutually related points of view. *Features* are abstractions from the requirements set on the product family. A feature may include a number of *subfeatures* as its constituent elements, and be characterised by a number of *attributes*. The technical aspects of the product family are captured by its *architecture* that is defined in terms of *components*. A component may have a number of other components as its *parts*, and be characterised by a number of attributes. Further, the possibilities for interaction of a component are specified by its *interfaces* and *bindings* between them.

Kumbang Configurator supports the user in the configuration task by providing a graphical user interface through which the user can enter his specific requirements for a product individual. Further, the configurator checks the configuration for *consistency* and *completeness* after each step, and deduces the consequences of the selections made so far. The necessary inferences are implemented using *smodels* [Simons *et al.*, 2002], which is a general-purpose inference tool based the stable model semantics of logic programs.

To the best of our knowledge, a tool supporting modelling concepts similar to Kumbang has not been previously implemented. Hence, Kumbang Configurator is the main contribution of this paper.

The remainder of the paper is organised as follows. Section 2 discusses Kumbang language. Section 3 presents Kumbang Configurator. Further, Section 4 discusses how Kum-

bang Configurator has been validated with example cases. Thereafter, Section 5 compares Kumbang Configurator with related work. Finally, Section 6 draws conclusions and suggestions for future work.

2 Kumbang Language

In this section, we discuss the Kumbang language and its background. First, we will iterate on *feature modelling* and Forfamel, the feature modelling method in Kumbang. Second, we will give similar treatment to architecture description and Koalish. Finally, we will discuss how Forfamel and Kumbang are integrated in Kumbang.

Feature modelling has become a popular method for modelling requirements of software product families [Kang *et al.*, 1990; Czarnecki and Eisenecker, 2000]. A feature has been defined as a characteristic of a system that is visible to the end-user [Kang *et al.*, 1990], and as a logical unit of behaviour that is specified by a set of functional and quality requirements [Bosch, 2000]. Feature modelling methods are based on organising features into *feature models* that typically take the form of a tree. Such a tree captures the variability of the software product family modelled in terms of its features.

Forfamel [Asikainen, 2004] is a feature modelling method that synthesises existing feature modelling methods with configuration modelling concepts stemming from the product configuration domain, more specifically from [Soininen *et al.*, 1998]. The fundamental modelling element of Forfamel is *feature type*; each feature type intentionally defines the properties of its instances, i.e., *features*. A feature type may include a *subfeature* definition that specifies the number and types of possible feature instances, thus defining the compositional structure of features. Further, a feature type may define *attributes* that characterise its instances. The value range for an attribute instance is attained by defining *attribute value types*. Moreover, a feature type may be defined one or more *supertypes*; a feature type inherits the property definitions of its supertypes. Finally, *constraints* between different combinations of features may be stated, thus restricting the valid combinations of features.

Example. In order to clarify the concepts presented in this paper, we provide a running example. The configuration model in Figure 1 depicts a small client-server system with varying number of clients. The model contains only one feature type *RootFeature*. However, feature type *RootFeature* contains two attribute definitions. Attribute *numberOfClients* is of type *Int2*, which means that its value can be either one or two. Attribute *isExtended* is of type *Boolean* and thus its value can be either *yes* or *no*. □

However, features as are such not sufficient for describing all the relevant aspects of software product families. In more detail, means for describing the technical aspects of the product family are needed. Towards this end, Koalish [Asikainen *et al.*, 2003; Asikainen, 2004] is a method for modelling *software product family architectures*. In other words, Koalish can be used to describe the overall structure of the software product family in terms of its architectural elements. Koalish is based on Koala [van Ommering *et al.*, 2000], a component model and an architecture description language de-

```
Kumbang model ClientServer
  root feature RootFeature
  root component RootComponent

feature type RootFeature {
  attributes
    Int2 numberOfClients;
    Boolean isExtended;
  implementation
    cardinality($.client) = value(numberOfClients);
    value(isExtended) = yes <=>
      for_all(X : $.client) instance_of(X, ExtendedClient);
}

component type RootComponent {
  contains
    (Client, ExtendedClient) client[1-2];
    Server server;
}

component type Client {
  requires RemoteProtocol caller;
}

component type ExtendedClient {
  requires RemoteProtocol caller;
}

component type Server {
  provides RemoteProtocol callee { grounded };
}

interface type RemoteProtocol {
  sendData; checkStatus;
}

attribute type Boolean = { yes, no }
attribute type Int2    = { 1, 2 }
```

Figure 1: A sample Kumbang model that describes a small client-server system.

veloped and used at Philips Consumer Electronics. Koalish shares its conceptual basis with Koala, but adds several variability mechanisms familiar from the product configuration domain, see e.g. [Soininen *et al.*, 1998]. In more detail, a Koalish model can contain *component* types that are instantiated as component instances. A component type can define the number and types of components as parts under the corresponding component instance. Further, a component type can define the type and direction of *interface* instances contained in the component instance. A *required* interface signals that the functions enlisted in its interface type must be implemented by a *provided* interface with such functions. In order to satisfy required interfaces, interface instances can be *bound* with each other. Also, similarly as in Forfamel, constraints concerning the combinations of architectural elements may be stated. Finally, in a manner similar to Forfamel attribute mechanism, attributes can be used for characterising properties of component instances.

Example. The running example defines a two-level compositional structure for component instances. Namely, the root component type *RootComponent* defines parts *server* and *client*. Part definition *server* states that a *RootComponent* instance must contain one component of type *Server*, while part definition *client* states that a *RootComponent* instance must contain one or two *Client* or *ExtendedClient* components. Further, due to the interface definition *caller*, a component of type *Client* or *ExtendedClient* contains one required interface of type *RemoteProtocol*. In a similar manner, a *Server*

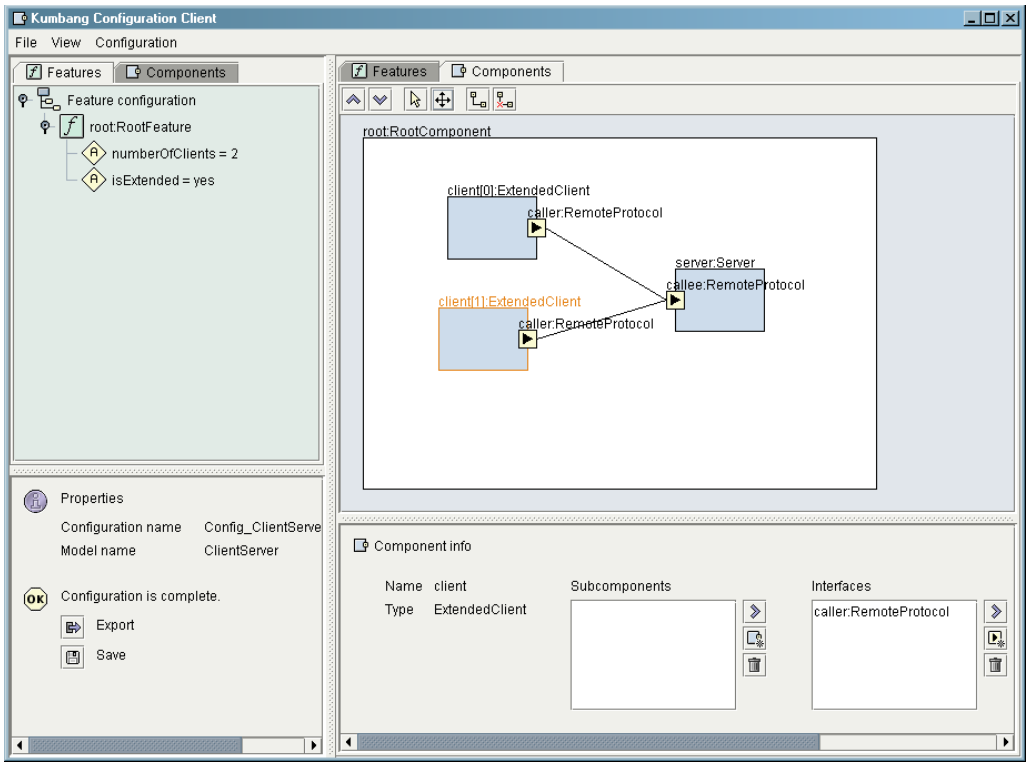


Figure 2: A screenshot from Kumbang Configurator user interface. This screenshot illustrates how the configuration model presented in our running example can be configured.

component contains a provided interface of type *RemoteProtocol*. Interface type *RemoteProtocol* consists of two functions, *sendData* and *checkStatus*. □

The Kumbang language combines Forfamel and Koalish into a single modelling language for configurable software product families. Hence, Kumbang enables modelling a software product family simultaneously from a feature and architectural point of view. However, these two views are often related to each other, just like system requirements and architecture are related. Therefore, Kumbang enables specifying how features are *implemented* by architectural entities. In more detail, feature types may include *implementation constraints* that must hold for the architecture in order for the product individual to provide the specific feature. A model of a configurable software product family presented in Kumbang is a *configuration model*, in the sense of [Soininen *et al.*, 1998], of the product family.

Example. The configuration model in our running example is a Kumbang model, and it exemplifies how features and components can be related to each other. The implementation constraints defined in feature type *RootFeature* relate selected attribute values to component configuration. The value of attribute *numberOfClients* is related to the number of client components as parts under *RootComponent*, while value *yes* for attribute *isExtended* is equivalent to all client components being of type *ExtendedClient*. □

3 Kumbang Configurator

This section provides an overview of Kumbang Configurator. Subsection 3.1 presents a brief overview of the functionality provided by the system, while Subsection 3.2 discusses the system architecture. Finally Subsection 3.3 discusses how configuration reasoning has been implemented.

3.1 Provided Functionality

As discussed earlier, Kumbang Configurator is based on Kumbang language; thus it can be used for deriving configurations that contain both features and components. However, the tool also supports Forfamel and Koalish models (see Section 2) as special cases. This means that the tool can be used for deriving configurations with features only, or configurations with components only.

Kumbang Configurator takes a configuration model of a software product family represented in Kumbang as input. The configuration model is used as the basis of the configuration task. The configurator offers a graphical user interface (see Figure 2) through which the user can make *selections* that modify the configuration. In more detail, selections can add or remove features, components, interfaces or bindings and set attribute values.

The graphical user interface provided by Kumbang Configurator visualises the configuration in a way that resembles existing notations known in the software product family community. For example, the visualisation of the component con-

figuration (right side of Figure 2) resembles the graphical notation of Koala [van Ommering *et al.*, 2000]. The user interface has been designed to show all available selections explicitly. Thus the tool can be used without additional in-depth knowledge of the configuration model.

Kumbang Configurator checks whether the configuration is *consistent* and *complete*. A consistent configuration is such that no rules of the configuration model have been violated. In contrast, a complete configuration is such that all necessary selections have been made. Further, Kumbang Configurator deduces the direct consequences of the configuration selections made so far. This means that the tool can automatically add selections implied by previous selections, identify selections conflicting with previous selections, and illustrate these selections in the user interface. Especially, selections in the feature hierarchy may have implications on the architecture, as specified by the implementation constraints in feature types. Consequently, the configuration model may be such that the architecture of the individual is completely deduced from the selections made in the feature hierarchy. This situation corresponds to the typical assumption that requirements are used as a basis for specifying software architecture. On the other hand, the constraints may take the form of equivalence constraints, implying that selections about the architecture may have implications in the feature hierarchy.

Finally, when the configuration is complete, the user of the tool can request a description of the product individual. This description can be used as a basis for building the product from existing software assets. However, Kumbang Configurator does not itself build the product; a separate builder is needed for this purpose.

Example. Figure 2 shows a screenshot from Kumbang Configurator applied to the running example. The current feature configuration is shown on the left as a tree, while a diagram of the architectural configuration is shown on the right. The user of the tool has set attribute *isExtended* to value *yes* and attribute *numberOfClients* to *2*. Based on these two selections, Kumbang Configurator has automatically deduced the entire component configuration. In the component configuration, root component *RootComponent* is composed of two components of type *ExtendedClient* and one of type *Server*. The interfaces of these components have been bound accordingly. □

3.2 System Architecture

Kumbang Configurator follows the distributed client-server architectural style (see Figure 3). The communication between the client and the server happens through Java Remote Method Invocation (RMI). In particular, the server provides a RMI interface for clients. A server may serve multiple clients simultaneously.

The client is implemented in the Java programming language. It provides a graphical user interface and interacts with the user of the tool during the configuration task.

The server is likewise implemented in the Java programming language. It takes care of the configuration reasoning (see Subection 3.3). Since this reasoning utilises *smodels* module, and since *smodels* has been implemented in C++, the server has to connect to *smodels* module through Java Native Interface (JNI).

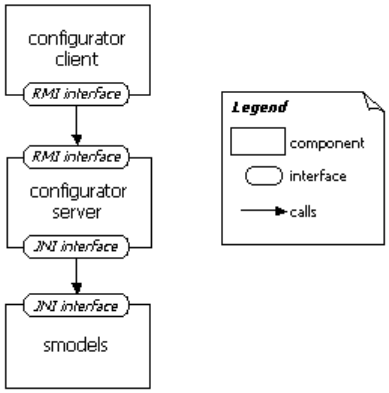


Figure 3: Kumbang Configurator follows the distributed client-server architectural style.

There are several reasons for following the client-server style. Firstly, as configuration reasoning can be computationally expensive, the server can run on a dedicated machine. Secondly, the server provides centralised model management. The server stores configuration models in a repository that is accessed by clients. Centralised model management enables separation between domain engineering and application engineering activities: those who perform the configuration task (for example, sales personnel) can use configuration models directly from the repository.

3.3 Configuration Reasoning

The term *configuration reasoning* refers to the inferences required to implement the configurator. In more detail, this includes checking configurations for consistency and completeness with respect to a configuration model and deducing the consequences of the selections made.

To implement the reasoning, the Kumbang model is translated to a form of logic programs; the translation is illustrated in the upper part of Figure 4. In more detail, the Kumbang model is first translated into Weight Constraint Rule

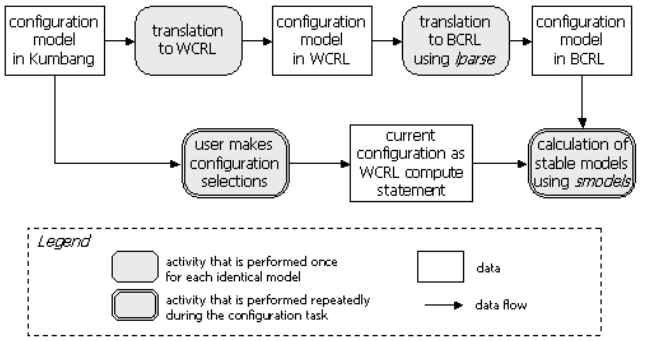


Figure 4: How Kumbang Configurator uses *smodels* for configuration reasoning.

Language (WCRL) [Simons *et al.*, 2002], a general-purpose knowledge representation language; details of this translation can be found in [Asikainen, 2004]. The resulting WCRL program is further translated into a more restricted form of weight constraint rules, namely Basic Constraint Rule Language (BCRL). This latter translation is carried out by the *lparse* module of the *smodels* system [Simons *et al.*, 2002]. It should be noted that the translation from WCRL to BCRL can be time-consuming, but it only needs to be done after a configuration model is created or changed.

Example. Before the configuration task begins, Kumbang Configurator translates the running example to WCRL. The resulting WCRL model contains 110 weight constraint rules that specify the configuration model. For example, a rule

$$1\ \{\ \mathit{hasattr}(\ X, \mathit{isExtended}, V\ ) : \mathit{attrBoolean}(V)\ \}\ 1$$
$$:\text{-}\ \mathit{in}(X), \mathit{instance}(\ X, \mathit{featRootFeature}\ ).$$

states that if an instance *X* of type *RootFeature* is currently in the configuration, it must have exactly one attribute named *isExtended*, and this attribute must have a value *V* from the range specified by attribute value type *Boolean*. □

The inference steps during the configuration task are illustrated in the lower part of Figure 4. In short, the selections made by the user are translated into *compute statements* that can be combined with the BCRL representation of the configuration model. Using this combination, the *smodels* [Simons *et al.*, 2002] system can be used to compute the consequences, which are in turn fed back in the client and represented in the user interface.

For further information on this topic, please refer to [Myllärniemi, 2005; Asikainen, 2004].

4 Validation

Kumbang Configurator has been validated using two sample cases. The first case represents a significant portion of real life case, and the second is an invented toy example that demonstrates additional aspects of the tool.

The first case originates from Robert Bosch GmbH [MacGregor, 2004]. Robert Bosch GmbH is a company developing various automotive systems containing embedded software. A distinguishing characteristic of automotive industry is the large number of variants. The case used for validation is a part of a model for a car periphery system (CPS), and it has been obtained from a presentation by John MacGregor [MacGregor, 2004]. The original presentation included a demonstration that showed how this particular system has been configured. The configurator tool used in the demonstration has been presented in [Hotz *et al.*, 2004].

However, the first case does not utilise interfaces or bindings. Since these are major contributions of Kumbang Configurator, another case was constructed for this purpose. Although the details of the case were invented for evaluation purposes, the case was motivated by a real-life system. The case describes a distributed weather station network that is used for measuring various weather-related quantities at multiple physical locations.

The authors modelled and configured both cases. Modelling included writing the cases into a Kumbang configuration model. As a result, the first case yielded a configuration model with dozens of feature, component, and attribute types, while the second configuration model was somewhat smaller. The configuration task included deriving various different configurations using Kumbang Configurator. Although systematic performance tests have not yet been run, these cases indicate that the system performance is adequate. (For example, checking the configuration state for the weather station configuration model takes approximately 60ms on a 750MHz Pentium PC.) However, it is yet unknown how Kumbang Configurator can handle very large configurations.

5 Related Work

There exists a number of configurator tools that have been designed to support configuration task. However, only few have been designed for the software domain. To some extent, it is possible to use a configurator designed for traditional mechanical products for configuring software [Asikainen *et al.*, 2004]. However, to our knowledge, none of the existing configurators currently provides the same set of functionality as Kumbang Configurator.

WeCoTin [Tiihonen *et al.*, 2003] is a configurator designed mainly for traditional, mechanical products. It employs many techniques that are also used in Kumbang Configurator, e.g., it uses *smodels* to implement the necessary reasoning. However, there are several differences between WeCoTin and Kumbang Configurator. First and foremost, WeCoTin doesn't recognise connectors or interfaces, which are an essential part of Kumbang Configurator. Secondly, WeCoTin lacks the separation between features and architectural elements. It can be used for modelling both of them, but not in parallel in a same configuration model.

There are a few configurator tools that have been designed for software domain. One of them is presented in [Hotz *et al.*, 2004]; it is built on top of existing product configurators. When comparing the approach in [Hotz *et al.*, 2004] with Kumbang Configurator, one can find several similarities. Both approaches support the separation between features and components, and both provide many similar variability mechanisms. However, there are also several differences. For example, Kumbang Configurator supports connectors and interfaces, whereas [Hotz *et al.*, 2004] does not.

Further, there are software configuration tools that do not originate from the field of configurable product research. Mae [Roshandel *et al.*, 2004] is a system that combines architectural description languages (ADL) with software configuration management (SCM) principles. Due to the fact that both Mae and Kumbang Configurator bring variability mechanisms to ADLs, the architecture-related modelling concepts in these two approaches bear many similarities. However, Mae provides several capabilities, such as *evolution* and *anytime variability*, which are lacking from Kumbang Configurator. In contrast, Mae does not include features. Further, the configuration reasoning mechanisms in Mae are rather limited compared to Kumbang Configurator; for example, validity checking is performed only after the configuration has been constructed.

6 Conclusions and Future Work

This paper presented Kumbang Configurator, which is a prototype tool for configuring product individuals from configurable software product families. Kumbang Configurator is based on modelling language Kumbang, which combines feature-based and architecture-based modelling methods. Thus Kumbang language has been designed for software product family domain. Further, Kumbang Configurator utilises existing inference engine *smodels* for configuration reasoning.

However, we have identified areas that require further research.

In order to ease the modelling task, we need to provide a graphical modelling tool that enables easy creation of Kumbang configuration models. We are currently working on such modelling tool. Further, Kumbang Configurator does not currently support activities after the configuration task, that is, building the source code into an executable artifact. We are investigating on how to relate architecture to actual implementation entities, and how to provide tool support for the build task.

Finally, further empirical knowledge is needed on the performance of Kumbang Configurator. For this purpose, systematic empirical tests are needed. It would be especially interesting to study how connectors affect the performance of the system.

References

[Asikainen *et al.*, 2003] T. Asikainen, T. Soininen, and T. Männistö. A Koala-based ontology for configurable software product families. In *IJCAI 2003 Configuration workshop*, 2003.

[Asikainen *et al.*, 2004] Timo Asikainen, Tomi Männistö, and Timo Soininen. Using a configurator for modelling and configuring software product lines based on feature models. In *Proceedings of the Workshop on Software Variability Management for Product Derivation, at Software Product Line Conference (SPLC3)*, 2004.

[Asikainen, 2004] Timo Asikainen. *Modelling Methods for Managing Variability of Configurable Software Product Families*. Licentiate thesis, Helsinki University of Technology, 2004.

[Bosch, 2000] Jan Bosch. *Design and Use of Software Architectures: Adapting and Evolving a Product-Line Approach*. Addison-Wesley, Boston, 2000.

[Bosch, 2002] Jan Bosch. Maturity and evolution in software product lines: Approaches, artefacts and organization. In Gary J. Chastek, editor, *Proceedings of the Second Software Product Line Conference (SPLC2)*, pages 257–271, 2002.

[Clements and Northrop, 2001] Paul Clements and Linda Northrop. *Software Product Lines—Practices and Patterns*. Addison-Wesley, Boston, 2001.

[Czarnecki and Eisenecker, 2000] K. Czarnecki and U.W. Eisenecker. *Generative Programming*. Addison-Wesley, Boston, 2000.

[Hotz *et al.*, 2004] Lothar Hotz, Thorsten Krebs, and Katharina Wolter. Combining software product lines and structure-based configuration—methods and experiences. In *Proceedings of the Workshop on Software Variability Management for Product Derivation, at Software Product Line Conference (SPLC3)*, 2004.

[Kang *et al.*, 1990] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, ADA 235785, Software Engineering Institute, 1990.

[MacGregor, 2004] John MacGregor. CONIPF—configuration in industrial product families. Presentation in Workshop on Software Variability Management for Product Derivation, at Software Product Line Conference (SPLC3), 2004.

[Myllärniemi, 2005] Varvana Myllärniemi. Kumbang Configurator—a tool for configuring software product families. Master's thesis, Helsinki University of Technology, Department of Computer Science and Engineering, 2005.

[Raatikainen *et al.*, 2005] Mikko Raatikainen, Timo Soininen, Tomi Männistö, and Antti Mattila. Characterizing configurable software product families and their derivation. *Software Process: Improvement and Practice*, 10(1), 2005.

[Roshandel *et al.*, 2004] Roshanak Roshandel, Andre van der Hoek, Marija Mikic-Rakic, and Nenad Medvidovic. Mae—a system model and environment for managing architectural evolution. *ACM Transactions on Software Engineering and Methodology*, 18(2):240–276, 2004.

[Simons *et al.*, 2002] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138:181–234, 2002.

[Soininen *et al.*, 1998] T. Soininen, J. Tiihonen, T. Männistö, and R. Sulonen. Towards a general ontology of configuration. *AI EDAM (Artificial Intelligence for Engineering Design, Analysis and Manufacturing)*, 12(4):357–372, 1998.

[Tiihonen *et al.*, 2003] Juha Tiihonen, Timo Soininen, Ilkka Niemelä, and Reijo Sulonen. A practical tool for mass-customising configurable products. In *Proceedings of the 14th International Conference on Engineering Design (ICED'03)*, 2003.

[van Ommering *et al.*, 2000] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, 2000.

PLM-integrated Configurators for Machine and Plant Construction

Dr. Philipp Ackermann
Perspectix AG
Hardturmstrasse 169, CH-8005 Zürich
ackermann@perspectix.com

Abstract

Product configurators in sales and project engineering for machine and plant construction have to deal with geometric as well as logistic constraints. Using 3D product representations and multi-structuring logic enables the consistent transformation between constructive BOMs for CAD systems and manufacturing BOMs for ERP systems.

1 Product Lifecycle Management

Product innovation is no longer a genius act of an autonomous developer team but more and more a simultaneous process crossing different departments and even company borders. The role of an engineer shifts from pure creative work on technical aspects to administrative and communicative tasks because he has to take into account and is influenced by requirements from planning, production, procurement, marketing and service processes [Stark, 2005]. Product information becomes a corporate resource within a collaborative product development process. Due to shorter development cycles and global collaboration, a demanding pressure exists to share knowledge and to make product information explicit and immediately available to others by digital means.

1.1 Product Systematics

While engineering-oriented Product Data Management (PDM) deals with CAD file repositories, version control, and engineering collaboration, today the focus shifts to enterprise-wide Product Lifecycle Management (PLM) including portfolio management, product architecture (standardization, modularization, platform strategy), variant and configuration management, pricing, cost and margin calculations, etc. PLM is therefore more than engineering support. The integration of diverse aspects on product information needs a unified corporate view and a common methodology on managing product knowledge.

1.2 Visual Simulation in Product Configurators

A product knowledge base has to combine the table-based, commercial product data from the ERP system with the geometric-physical product data from CAD systems and has to model the interdependencies of the involved entities into a dynamic simulation environment. What has been successful in CAD systems – using visual front-ends to manage the complexity of the digital product model – needs to be transformed into visual interaction metaphors in sales, planning and service tools. The recent development in ERP user interfaces was concentrated in web-enabling existing dialog systems based on tables and forms. The next step will be a more fundamental shift by providing visual front-ends to the ERP and PLM backbone that transform the huge amount of data into interactive simulation worlds.

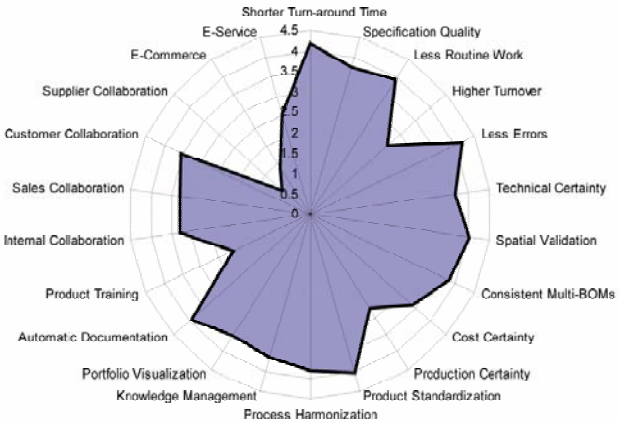


Figure 1: Benefit expectations in sales tools

1.3 Sales Engineering

Inspired by the survey in [Edwards and Riis, 2004], interviews with 15 Swiss and German manufacturers were conducted in the discrete industry on benefit expectations in IT-based sales engineering tools. The survey (Fig. 1, [Dörflinger and Ackermann, 2005]) has shown a large potential on improvements in comparison to traditional CRM approaches.

The main challenges in marketing of complex investment goods consists of

- custom-specific product and offer configuration,
- technical dimensioning and layout planning,
- collaborative project execution, and
- consistent BOM and document generation.

In global competition, machine and plant manufacturers are typically forced to provide basic engineering tasks for free in the acquisition phase. Due to increasing market pressure, technical and financial risk reductions are fundamental and raise the effort needed in sales engineering. Manufacturers are therefore shifting from unprofitable design-to-order models to individually configurable product systems based on reusable modules and assemble-to-order fulfillment.

2 PLM-integrated Tools for Technical Sales

Effective sales support has to bridge technical and logistical aspects, has to integrate CAD and ERP systems. By applying rule-based reasoning and constraint solving on logical as well as geometric associations, product configurators can form the foundation to keep product models consistent through their lifecycle along the key processes between development, production, and sales (Fig. 2).

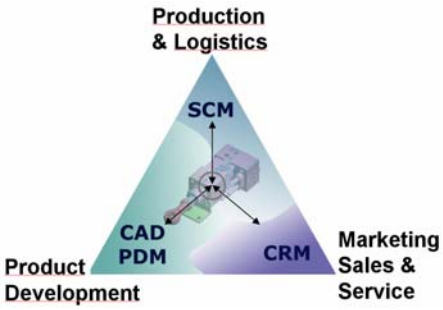


Figure 2: PLM integration

2.1 Spatial Composition

By reusing CAD data in the configurator, arrangements of modules can be validated on their compositional structure, on connected module interfaces (mating conditions), on distance constraints, and on collision detection. The product model of the P'X5 configurator (see www.perspectix.com) provides such spatial and geometric associations [Ackermann and Eichelberg, 2004].

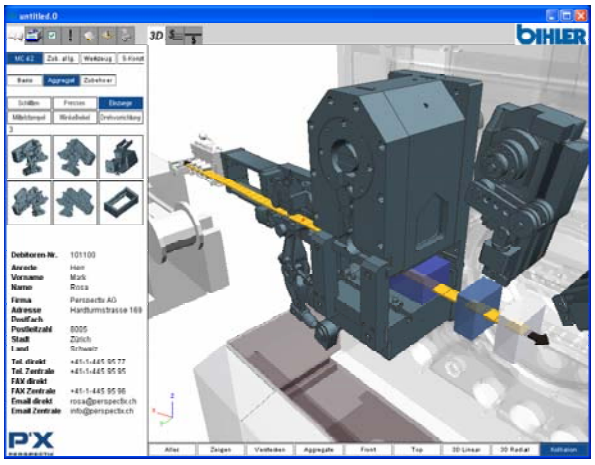


Figure 3: Geometric module composition of a machine

With in a catalog, the configurator provides to the user the sales and planning components, which may be composed interactively in 3D as "intelligent objects" knowing their potential positioning (Fig. 3).

2.2 Structural Aggregation

The connected components build the basis of structuring the module composition in a "middle-out" assembly approach (Fig. 4). Patterns of found component types and connected matings trigger the bottom-up generation of assemblies. In a top-down approach, single components may be resolved into their subparts. The combinatorics of the module composition and the structuring of assembly aggregations are then extended by the parametrics of property propagation and expression calculations both in their compositional and aggregational structure.

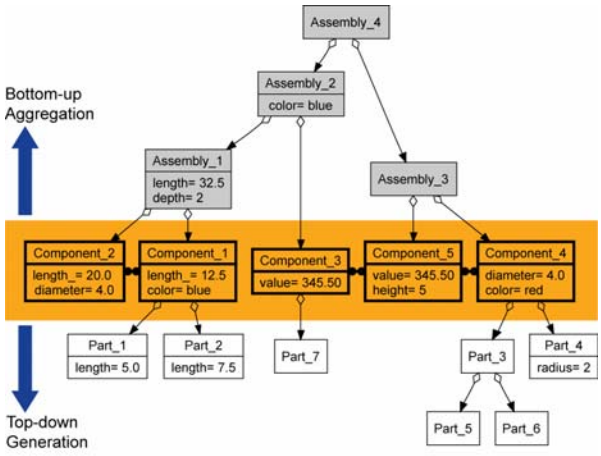


Figure 4: Structuring by a "middle-out" approach

2.3 Multi-Structering

An important aspect of PLM-integrated configurators is the ability to transform product structures between different views (Fig. 5). Not only the entities such as documents and product parts differ between views, but also the aggregational structures expressed as Bill Of Material (BOM).

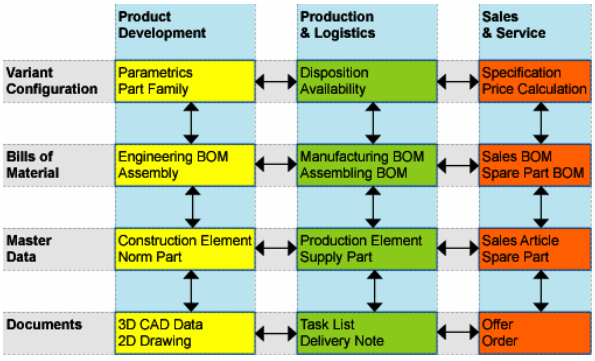


Figure 5: Transformation competence

By using rule-based automatism to generate different BOM structures in parallel, PLM-integrated configurators can provide BOM structures for the construction (CAD), production (PPS/ERP), logistics (SCM), and sales (CRM) and keep them all consistent. While Fig 4 represents the constructive building structure (sent to the CAD system), Fig.6 shows the commercial article list (sent to the ERP system) generated from the same module composition consisting of the same connected componenents.

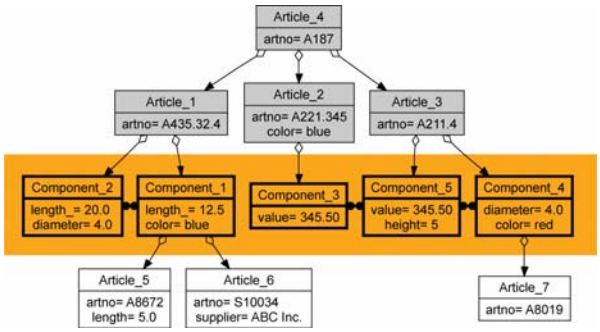


Figure 6: Structuring of a commercial sales BOM

View-specific BOM structures are the basis for view-specific document generation (Fig. 6), e.g., for sales offers, assembly lists, working plans, procurement orders, layout plans, process lists, etc.

3 Conclusion

Benefits of PLM-integrated configurators will come through the increase in the and efficiency of internal and external communication, through automation of routine work and bypassing of administrative procedures, by the decrease of errors, and the possibility of reusing product information along the value-chain of the whole product lifecycle.

References

[Stark, 2005] John Stark. *Product Lifecycle Management – 21st Century Paradigm for Product Realisation.* Springer, London, 2005.

[Ackermann and Eichelberg, 2004] Philipp Ackermann and Dominik Eichelberg. *Product Knowledge Management.* PETO Conference, Technical University of Denmark, Copenhagen, 2004.

[Edwards and Riis, 2004] K. Edwards and J. Riis. *Expected and Realized Costs and Benefits when Implementing Product Configuration Systems.* Proceedings of the 8th International Design Conference 2004, p. 183-196, Design Society, Dubrovnik, 2004.

[Dörflinger and Ackermann, 2005] Markus Dörflinger, Philipp Ackermann. *CRM im Maschinen- und Anlagenbau.* SMM, Schweizer Maschinenmarkt, Zürich, 2005.

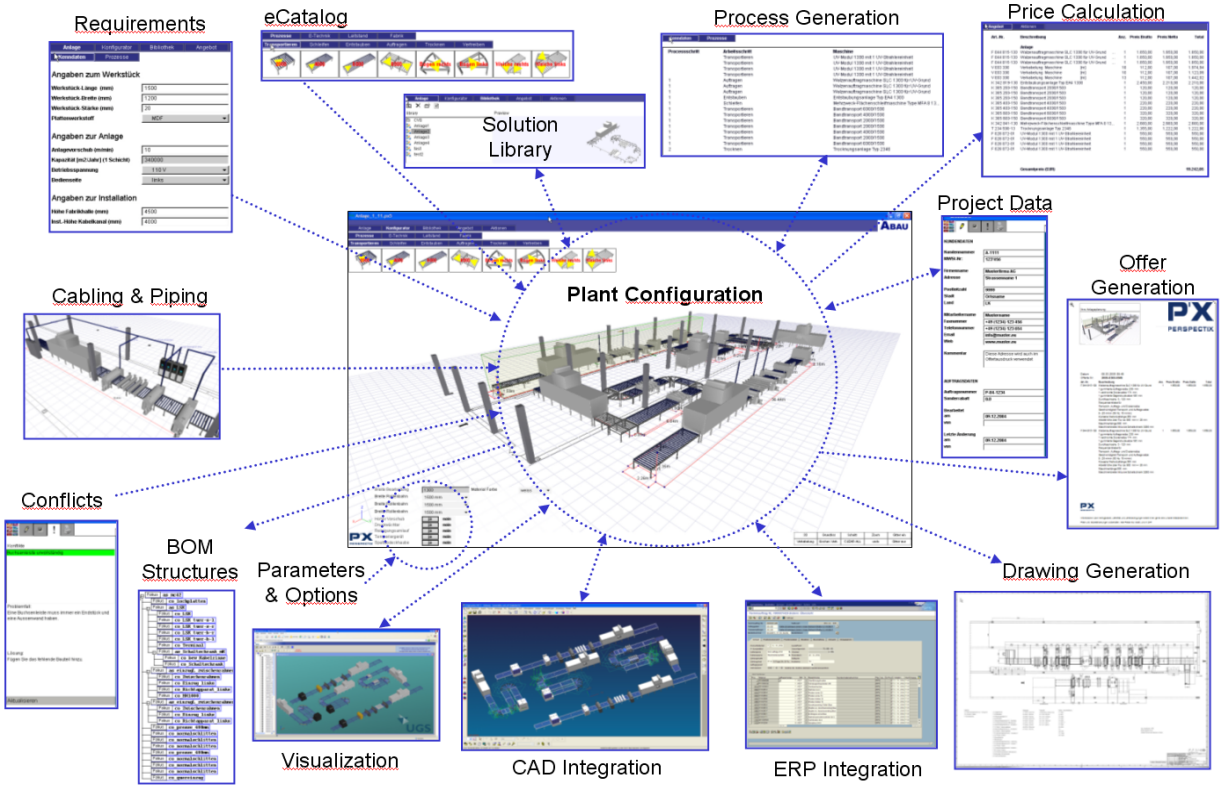


Figure 6: Configuration tasks for the basic engineering of a plant in the sales phase

Co-Configuration of Products and On-Line Service Manuals

Carsten Sinz and **Wolfgang Küchlin**
Steinbeis Technology Transfer Center OIT
WSI for Computer Science, University of Tübingen
Sand 13, 72076 Tübingen, Germany
http://www-sr.informatik.uni-tuebingen.de
{sinz, kuechlin}@informatik.uni-tuebingen.de

Abstract

As products are growing more complex, so is their documentation. With an increasing number of product options, the diversity in service and maintenance procedures grows accordingly. Different variants of a model line consist of different components and thus require different service procedures. SIEMENS Medical Solutions has thus decided, instead of having one common on-line service handbook for all its Magnetic Resonance Tomographs, to fragment the on-line documentation into small (help) packages, out of which a suitable subset is selected for each individual product instance. Selection of help packages is controlled by XML terms encoding Boolean choice conditions. To assure that the set of available help packages is sufficient for all valid product instances, we developed a tool called *HelpChecker* that uses SAT- and BDD-based methods to guarantee completeness of the on-line documentation and to support authors in finding residual gaps.

1 Introduction

Complex products require complex handbooks. Unfortunately, product manuals traditionally encompass not only those particular features the customer ordered, but all possible components of the whole model line. Over the last years, many products (e.g. cars, computers, telecommunication equipment, medical devices) have seen a tremendous increase in the number of available product options, the handbooks grew in size accordingly, thus making matters even worse.

Ideally, each product instance would be equipped with its own product manual, covering only the functionality that is in fact relevant for that product instance. However, this approach is still not very common, perhaps due to the fact that it would require a configuration of the manuals themselves in conjunction with the configuration of the product.

SIEMENS Medical Solutions recently introduced a new XML-based product configuration system for their Magnetic Resonance (MR) Tomographs. In the course of this introduction, they decided to equip their products with individually configured on-line service manuals. Now, the material of all handbooks is split up into smaller help packages, each covering a clearly delimited topic. Each service handbook is then automatically generated as a suitable selection of those packages. Selection of packages is controlled by Boolean logic formulae encoded as XML terms, and during configuration of a product instance its manual is configured simultaneously.

The authors of the help packages can decide autonomously about how to break down the whole help documentation into smaller packages. So it is their own decision whether to write a whole bunch of smaller packages, one for each system configuration, or to integrate similar packages into one. Writing of service manuals does not follow the configuration structure of each individual system but is done for similar service situations across all product lines. However, this approach makes it hard to determine whether all needed help packages already have been written or whether there are cases that still need to be covered. To guarantee completeness of the manuals for all possible, valid product instances a final cross-check is needed. We therefore developed a tool called *HelpChecker*, which informs the authors of the documentation department which parts of the manual they still have to write.

In the following, we describe the method used by SIEMENS for documenting their MR products, how service manuals are associated with product instances, and how we check the set of all help packages for completeness with our add-on tool *HelpChecker*.

2 Configuring SIEMENS MR Tomographs

Many different formalisms have been proposed to model the structure of complex products [Mittal and Frayman, 1989; Sabin and Weigel, 1998; Soininen *et al.*, 1998; McGuiness and Wright, 1998; Küchlin and Sinz, 2000]. The method used by SIEMENS for the configuration of their MR systems was developed in collaboration with the first author of this paper and resembles the approach presented by Soininen *et al.* [Soininen *et al.*, 1998]. Structural information is explicitly represented as a tree. This tree serves two purposes: first, it reflects the hierarchical assembly of the device, i.e. it shows the constituent components of larger (sub-)assemblies; and, second, it collects all available, functionally equivalent configuration options for a certain functionality. These two distinct purposes are reflected by two different kinds of nodes in the tree, as can be seen from the example in Fig. 1: *Type Nodes* represent configurable entities and are employed to

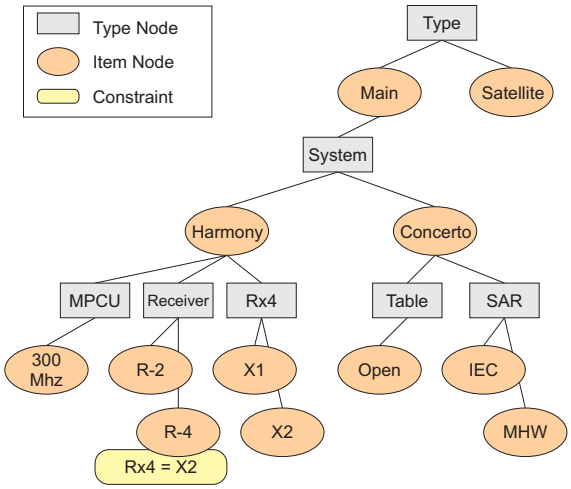


Figure 1: Product Structure of Magnetic Resonance Tomographs (Simplified Example).

reflect the hierarchical structure of the product; *Item Nodes* represent individual configuration options. The two types of nodes appear strictly alternatingly on each path of the tree. All of a Type Node's children thus are Item Nodes and vice versa. Each Item Node is one configuration possibility for the parent node.

From the example tree shown in Fig. 1 we may, e.g., conclude that there are two alternatives for choosing a *System*: *Harmony* and *Concerto*. A *Harmony* system possesses three configurable (direct) subcomponents, of type *MPCU*, *Receiver*, and *Rx4*, respectively. The receiver, in turn, may be selected from the two alternatives *R-2* and *R-4*. Choosing the latter option puts an additional restriction on the configurable component *Rx4*: this has to be selected in its form *X2*. Each type node possesses additional attributes *MinOccurs* and *MaxOccurs* to bound the number of subitems of the respective type to admissible values. Assuming that for each type exactly one item has to be selected (i.e. *MinOccurs* = *MaxOccurs* = 1 for all type nodes), the configuration tree shown in Fig. 1 permits, e.g., the following valid configuration (set of assignments):

$$\begin{array}{ll} \text{Type} = \text{Main} & \text{System} = \text{Harmony} \\ \text{MPCU} = \text{300MHz} & \text{Receiver} = \text{R-4} \\ \text{Rx4} = \text{X2} & \end{array}$$

3 Configuring Service Manuals

Within the SIEMENS system, the tree describing all product configurations is represented as an XML term. The term corresponding to the tree of Fig. 1 is shown in Fig. 4. All XML terms are checked for well-formedness using XML Schemas [XMLSchema, 2001]. Part of the XML Schema describing valid XML terms for product configurations is shown in Fig. 5.

The on-line help pages that are presented to the user of an MR system may depend on the configuration of the system. For example, help pages should only be offered for those components that are in fact present in the system configuration. Moreover, for certain service procedures (e.g., tune up, quality assurance), the pages depend not only on the system configuration at hand, but also on the (workflow) steps that the service personnel already has executed. Thus, the help system depends both on the configuration and the state of the workflow.

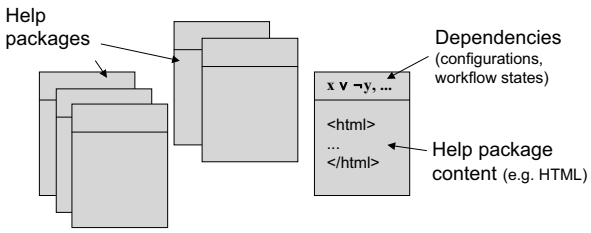


Figure 2: Illustration of Help Packages: For each system configuration and workflow state a suitable help package has to be selected (controlled by dependencies).

To avoid writing the complete on-line help from scratch for each possible system configuration and all possible workflow states, the whole help system is broken down into small *Help Packages* (see Fig. 2). A help package contains documents (text, pictures, demonstration videos) on a specialized topic. The authors of the help packages decide autonomously about how they break down the whole help document into smaller packages. So it is their own decision whether to write a whole bunch of smaller packages, one for each system configuration, or to integrate similar packages into one.

```
<Package ID="HLP_HP-1-181203-01-001" Name="HP-1-...">
   <Content> ... </Content>
   <Dependencies>
      <Dependency>
         <RefType IDREF="INT_Workflow">
            <RefItem IDREF="INI_Workflow_TUNEUP"/>
         </RefType>
         <RefType IDREF="INT_System">
            <RefItem IDREF="INI_System_003"/>
         </RefType>
      </Dependency>
   </Dependencies>
</Package>

<Context>
   <RefType IDREF="INT_System">
      <RefItem IDREF="INI_System_003"/>
   </RefType>
   <RefType IDREF="INT_Workflow">
      <RefItem IDREF="INI_Workflow_TUNEUP"/>
   </RefType>
   <RefType IDREF="INT_WorkflowMode">
      <RefItem IDREF="INI_WorkflowMode_General"/>
   </RefType>
   <RefType IDREF="INT_WorkflowSfp">
      <RefItem IDREF="INI_WorkflowSfp_SfpTuncalOpen"/>
   </RefType>
</Context>
```

Figure 3: Example of a Help Package (with Dependencies) and a Help Context.

Now, in order to specify the assignment of help packages to system configurations, a list of *dependencies* is attached

```
<Config auto-ns1:noNamespaceSchemaLocation="Config.xsd"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <Structure>
      <Type IDREF="INT_ConsoleType" MinOccurs="1" MaxOccurs="1">
         <Item IDREF="INI_ConsoleType_Sat"/>
         <Item IDREF="INI_ConsoleType_Main">
            <SubType IDREF="INT_System" MinOccurs="1" MaxOccurs="1">
               <!-- Harmony -->
               <Item IDREF="INI_System024">
                  <SubType IDREF="INT_Comp_MPCU" Default="INI_Comp_MPCU300" ReadOnly="true" MinOccurs="1" MaxOccurs="1">
                     <Item IDREF="INI_Comp_MPCU300"/>
                  </SubType>
                  <SubType IDREF="INT_Comp_RXNumOf" Default="INI_Comp_RXNumOf1" MinOccurs="1" MaxOccurs="1">
                     <Item IDREF="INI_Comp_RXNumOf1"/>
                     <Item IDREF="INI_Comp_RXNumOf2"/>
                  </SubType>
                  <SubType IDREF="INT_Comp_ReceiverNumOf" MinOccurs="1" MaxOccurs="1">
                     <Item IDREF="INI_Comp_ReceiverNumOf2"/>
                     <Item IDREF="INI_Comp_ReceiverNumOf4">
                        <Conditions>
                           <Condition Type="INT_Comp_RXNumOf" Op="eq" Value="INI_Comp_RXNumOf2"/>
                        </Conditions>
                     </Item>
                  </SubType>
               </Item>
               <!-- Concerto -->  ...
            </SubType>
         </Item>
      </Type>
   </Structure>
</Config>
```

Figure 4: Product Structure Tree of Fig.1 in XML Representation (Excerpts).

to each help package, in which the author lists the system configurations and workflow states for which his package is suitable (see Fig. 3 for an example): all of a dependency's `RefType`/`RefItem` assignments must match in order to activate the package and to include it into the set of on-line help pages for that system. Multiple matching situations may be specified by associating further `Dependency`-elements with the package.

The situations for which help packages must be available are specified by the engineering department using so-called *Help Contexts*. A help context determines system parameters and workflow steps for which a help package must be present. Examples for both a help package and a context (in XML representation) can be found in Fig. 3. The help package of this example fits any state of workflow *tune up* and all configurations of *System_003*. The example's context specifies that for step *TuncalOpen* in the *tune up* procedure for *System_003* a help package is required, in case the workflow mode is set to *General*.

Currently, almost a thousand help contexts have been defined for eleven MR systems (model lines), each with millions of different configuration possibilities. So, in spite of in-depth product knowledge, it is a difficult and time consuming task for the authors of help packages to find gaps (missing packages) or overlaps (ambiguities in package assignment) in the help system. To assist the authors, we therefore developed the *HelpChecker* tool, which is able to perform cross-checks between the set of valid system configurations, the situations for which help may be requested (determined by the contexts) and the situations for which help packages are available (determined by the packages' dependencies).

4 Checking Completeness and Consistency

Our implementation *HelpChecker* is a C++ program that builds upon Apache's Xerces XML parser to read the SIEMENS MR product structure and help package dependencies. From these data, it generates two propositional logic formulae:[1]

(A) **Completeness of the help system**. If this formula is valid, then for all valid MR configurations and all situations in which help may be requested, at least one matching help package is specified (via its dependencies).

(B) **No overlaps between help packages**. If this formula is satisfiable, there is a valid system configuration and a situation for which help may be requested (by a context) with more than one matching help package: an overlap exists (i.e. a fault in the help package assignment).

After having generated these formulae, they are checked for validity resp. satisfiability using BDD-based methods. In case of an error condition, a formula is generated describing the set of situations in which this error occurs. This formula, call it F, is simplified by existential abstraction over irrelevant variables using standard BDD techniques (i.e. by replacing F by $\exists x F$ or, equivalently, by $F|_{x=0} \vee F|_{x=1}$ for an irrelevant propositional variable x).

HelpChecker is embedded into a larger interactive authoring system for the writers of the help packages at SIEMENS. The authoring system offers a graphical user interface, by which the help packages' content and dependencies can be entered and maintained. The overall structure of the handbooks as well as the hyperlinks between different pages of

[1] Details on how these formulae are generated can be found in another publication [Sinz and Küchlin, 2004].

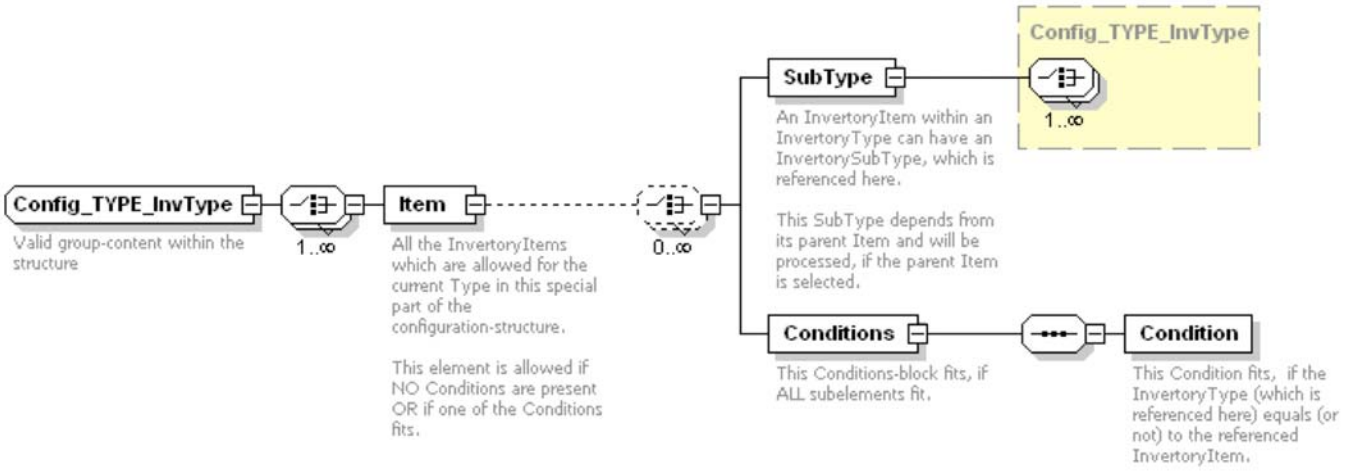


Figure 5: Graphical representation of the part of the XML-Schema that deals with the product structure.

the documentation are pre-specified by the technical department and cannot be altered by the help package authors. This also means that there is no intra-page hypertext navigation that is under control of the help checker. Versioning of products and handbooks is supported by the underlying XML data base schemas. In the authoring tool editors have to select a special version of the data they are working on, however. This also means that help packages for a follow-up version of the documentation may only be copied from an older version, but there is no way to specify (by constraints) that they are also valid for the new edition.

Production use of the system is just starting, thus we do not have user feedback yet, and can only report on manually created test cases (that were generated by SIEMENS documentation experts, though). These tests contained eleven basic MR systems, 964 help contexts and twelve (dummy) help packages. The BDDs reflecting test formulae (A) and (B) contained up to 9715 nodes and 458 variables (with intermediate BDD sizes during formula generation of over 100,000 nodes).[2] A complete check of our experimental XML help—including computation of error situations—took 6.96 seconds. The input data we used already contain the complete configuration structure for all of SIEMENS' current MR model lines. However, the number of help packages is much lower than what we expect during production use.

5 Conclusion

In this paper we have shown that individually configured service handbooks are feasible. By using a propositional logic encoding for both system configurations and the assignment of handbook chapters (help packages) to product instances, we were able to employ advanced Boolean logic techniques (like BDDs). Those turned out to have enough potential to handle the problems brought up by our completeness and consistency checks. Although we have demonstrated the feasibility of our method only for the MR systems of SIEMENS Medical Solutions, we suppose that the presented techniques are also usable for other complex products. More generally, we expect that a wide range of cross-checks between XML documents can be computed efficiently using propositional logic techniques.

[2] We have also conducted experiments using a SAT-Solver instead of the BDD package, and obtained very promising results.

References

[Küchlin and Sinz, 2000] W. Küchlin and C. Sinz. Proving consistency assertions for automotive product data management. *J. Automated Reasoning*, 24(1–2):145–163, February 2000.

[McGuiness and Wright, 1998] D.L. McGuiness and J.R. Wright. Conceptual modelling for configuration: A description logic-based approach. *AI EDAM*, 12(4):333–344, 1998.

[Mittal and Frayman, 1989] S. Mittal and F. Frayman. Towards a generic model of configuration tasks. In *Proc. of the 11th Intl. Joint Conf. on Artificial Intelligence*, pages 1395–1401, Detroit, MI, August 1989.

[Sabin and Weigel, 1998] D. Sabin and R. Weigel. Product configuration frameworks – a survey. *IEEE Intelligent Systems*, 13(4):42–49, July/August 1998.

[Sinz and Küchlin, 2004] Carsten Sinz and Wolfgang Küchlin. Verifying the on-line help system of SIEMENS magnetic resonance tomographs. In *Proc. of the 6th Intl. Conf. on Formal Engineering Methods (ICFEM'2004)*, pages 391–402, Seattle, WA, November 2004. Springer.

[Soininen *et al.*, 1998] T. Soininen, J. Tiihonen, T. Männistö, and R. Sulonen. Towards a general ontology of configuration. *AI EDAM*, 12(4):357–372, 1998.

[XMLSchema, 2001] *XML Schema Parts 0–2: Primer, Structures, Datatypes. W3C Recommendation*, May 2001.

Reconfiguration – A Problem in Search of Solutions

Peter Manhart
DaimlerChrysler AG
Research and Technology, REI/SP
Peter.Manhart@daimlerchrysler.com

Abstract

Many practical configuration problems are well understood in theory and solved in practice by existing product configurators. Yet, this does not apply to the reconfiguration of existing complex products: here, there exist neither theories nor systems that can deal with problems of realistic size.
In this paper we describe a reconfiguration task that originates in the complex nature of electric, electronic, and software components of vehicles. From this, we derive research questions in search of solutions.

1 Introduction

Reconfiguration is a serious challenge in industry and at least for complex products it is so far not solved in a feasible manner. This paper does not present possible solutions but aims at motivating for further research and developement efforts by presenting the application area vehicles.
Research on the product *configuration* domain is based on the notion of a configurable product: a product individual of a *configurable product* can be adapted to the requirements of a particular customer order. The possibilities for adapting the configurable product are predefined in a *configuration model* that explicitly and declaratively describes the set of legal product individuals. A specification of a product individual – a configuration - is produced in the configuration task based on the configuration model and a set of customer requirements. Efficient knowledge-based information systems - product configurators - have become an important and successful application of artificial intelligence techniques for companies selling products adapted to customer needs [(Faltings, B., Freuder, E. C. 1998)(Soininen, T., Stumptner, M., 2003)].

Reconfiguration on the other hand is an important task of after-sales. In reconfiguration, an existing product individual is modified to meet new requirements (Männistö et. al. 1999). Reconfiguration has always been a challenge, but the increased complexity and variability of electronic control units (ECU), as well as the virtual nature of their software and parameters has turned reconfiguration into a key issue in after-sales.

ECUs and software have various advantages. Thus most of the innovations of present vehicles, for example, automatic brake systems, come in conjunction with an electronic control unit and software. In vehicles ECUs and embedded software are strongly interwoven with the mechanical, electric, and electronic subsystems. But not only has a comparatively simple realization of complex add-on functionality increased the use of information technology in vehicles, the possibility to shift variability from hardware to software also make ECUs the solution of choice for engineers and manufacturers.
But are all of these advantages for nothing? State-of-the-art system development processes and system architectures induce a severe configuration complexity in products. The resulting challenges can be handled maturely in development by configuration management systems and in manufacturing by product configurators.
But somewhere along the way, a vehicle comes to a dealer's workshop and the owner needs a modification of a historical configuration that involves changes to the systems consisting of mechanical parts, ECUs, and software. Then the corresponding reconfiguration tasks have to be solved for the safety-critical system "vehicle" and, as our research has shown, reconfiguration is more complex than configuration.

In the next sections we will introduce our business object and the underlying markets rules, describe practical reconfiguration use cases, and motivate why reconfiguration is more complex than regular configuration issues.

2 The Business of Buses

In this section, we sketch the business environment of premium buses and illustrate what such buses look like from an architectural perspective.

2.1 The Business Environment

Customers buying high end buses do not buy off the shelf. Their own varying business models result in a high degree of individualization. Additionally, bus purchasers frequently impose customer-specific functionalities even in late manufacturing phases.
Especially the after-sales business is challenging. A bus owner recognizing a useful new function in a newly bought bus will often demand an upgrade of a five-year-old model

from the dealership. Hence configurations that were never a possible production state evolve.
Also, a bus can interact intensively with its environment by sending signals to its stopping places, reporting to the central public transportation computers, and controlling automated fueling stations. These third-party systems are not based on public standards but create a need for development of custom hardware and software interfaces.
All of these issues not only increase time pressure for the development of customer-specific embedded software but also boost variation dramatically. Additionally they have an impact on the system architecture and the software process. However, before we can address the concrete reconfiguration problem of buses as by these requirements, we need a basic understanding of the technical architecture of the system.

2.2 Electric Electronic (EE) Architecture of a Bus

Software-based functions in buses either reside in programmed controller units or are uploaded to programmable controllers.
A modern bus contains between four and six programmable controller units (see FPS in Figure 1). Controllers are connected by a low speed vehicle CAN (controller area network) and offer the following functions:

1. Gateway: They are communication gateways to programmed controllers at the aggregate level, e.g., to the electronic brake control unit.
2. IO: They convert hardware signals to CAN messages.
3. Bus functions: On the basis of a real-time operating system, they compute and control bus-specific functionality such as control of doors, climate, or audio/video equipment.

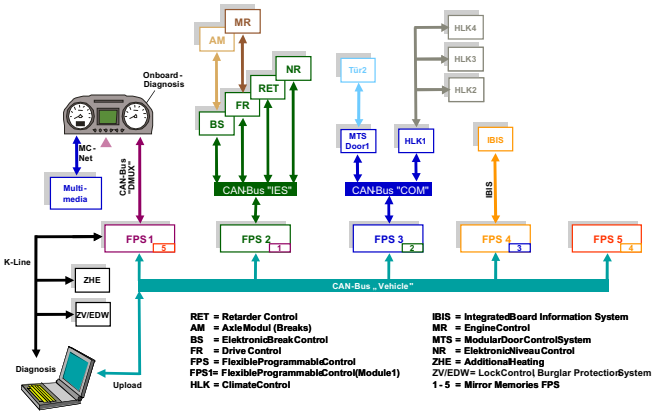


Figure 1: EE Architecture of a Premium Bus

Customer-specific software functionality is invariably implemented in the form of bus functions, in our case taking the form of so-called function plans (FUPs) according to IEC1131-3. When a customer requests a special functionality, a decision is made as to whether this will be fully or partially implemented in software. In this case, the engineering department has to produce one or more new function plans within time and quality constraints.

The conditions of engineering embedded software for high-end buses differ from those in many other automotive software engineering processes:

1. There is a higher degree of customer-specific functionality combined with greater time pressure and the necessity to deliver very high quality.
2. The resulting non-standard equipment, functions, or parameter sets are implemented by integrating different shares of third-party and OEM components.
3. Software for complex electric or electronic functions often runs on programmable computer controllers. In such a software environment, change *seems* easy from the customer's point of view.
4. Customers buying buses make a major investment (> 500,000 euros). They demand top quality. When new requirements come to mind, they expect them to be considered.

All of these issues increase product variation and, as a result, reconfiguration complexity significantly and is costly to manage.

2 The After-Sales Retrofitting Processes and the Reconfiguration Task

In the shop, the complexity and variability sketched in the last section accumulates to severe reconfiguration problems. We have two main business cases in mind:

1. Replacement of existing systems
   - Recall actions
   - Spare and wear parts
2. Enhancement of existing functionality
   - Using existing components
   - Engineering new customer-specific functions

Thus far, changing an existing complex system is rather a reengineering than a configuration task. In order to solve this problem more systematically or to more easily automate the retrofitting process, we have to become able to solve the reconfiguration task that can be defined informally as follows:

A *reconfiguration task* is solved if a starting configuration is transformed into a target configuration including all reconfiguration constraints.
The *reconfiguration constraints* include the following:

- The **use case**, e.g., replacing a part by a successor
- **Correctness** with respect to functional and non-functional requirements: basically, the new configuration should have a highest possible intersection with the requirements that is technically possible and affordable
- **Minimality** with respect to changing behavior and configuration items: no unnecessary part should be replaced and a minimum of behavior should be changed, if necessary at all

While these definitions are far from being formal definitions, they cover numerous practical requirements of after-sales retrofitting processes.

In the next section, we focus on the reasons why reconfiguration is more complex than configuration.

3 The Cause of the Problem

When traditional development engineers think about changing a configuration item their focus often lies on production. Here it is only necessary to model dependencies with respect to the currently released items. For reconfiguration in after-sales knowledge of dependencies to all historical versions of configurations items are potentially essential. The necessity of deriving and modeling these dependencies makes reconfiguration a significant more complex task than production-oriented configuration.

In Figure 2 the dashed line represents a dependency relation: Part 3, revision 2 needs part 1, revision 2 in order to function properly. For development purposes, it is not necessary to make these relations explicit as long as the new revision or variant of an existing part interworks with the other configuration items that are currently used. Also, production does not need them as they are able to switch their production lines at a certain point in time from the predecessor to the new versions. Basically, rule-based configuration approaches and configurators (Sinz, Kaiser Küchlein, 2003) reflect this mindset. Thus knowledge about interdependencies is often represented in a top-down constructive way: if a certain logical expression over requirement predicates holds true, then a certain set of configuration items is added to the configuration. Transversal dependencies between components are not necessary for production-oriented configuration and are therefore not modeled.

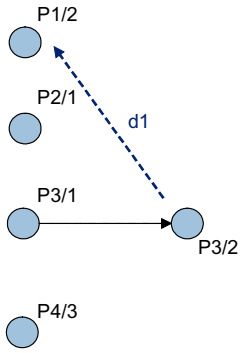


Figure 2: Developer View

Figure 3 sketches the situation from the after-sales perspective. The solid bold line represents the production configuration that enters the workshop, for example, with a defective ECU labeled P3/0. The currently available spare part is P3/2 (part 3, revision 2). The reconfiguration task includes answering the following question: What is a minimal correct target configuration that contains P3/2 instead of the defect P3/0?

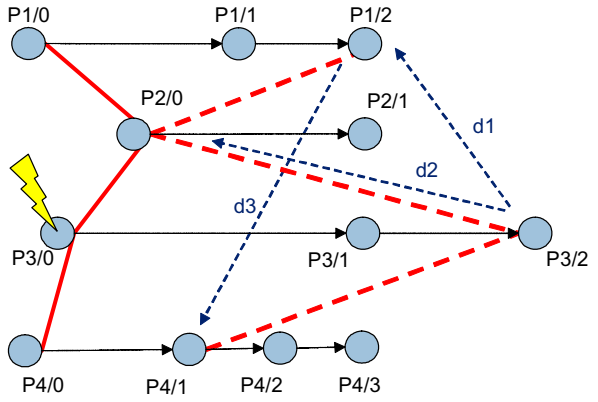


Figure 3: Aftersales View

In addition to the revision links, Figure 3 indicates the transversal dependencies d1, d2, and d3. Clearly, these have to be considered if the reconfiguration task is to succeed both correctly and minimally. In this example, the minimal and correct target configuration is represented by the dashed bold line. In addition to upgrading P3/0 to P3/2, d1 calls for an upgrade of P1/0 to P1/2, and via d3 this upgrade transitively requires upgrading P4/0 to P4/1. Together with thousands of other parts of the configuration, P2/0 can remain unchanged. In order to find this solution, all the dependencies (and non-dependencies) have to be considered as well as the corresponding compatibility relations. Unfortunately these dependencies are usually un-documented for a number of reasons. To cite just a few,

- they are complex to derive,
- they are large in size,
- they are not necessary to introduce product innovations.

However,

- they are also not covered by systematic approaches or configurators.

These are some of the reasons why reconfiguration of complex systems today is rather a reengineering than a configuration task. Whereas the first three issues are problem-immanent characteristics that come with complex systems, the last one can be recognized as a set of research challenges:

- How to enhance the production orientation by after-sales oriented thinking?
- What requirements engineering processes and product development processes lead to more reconfigurable products?
- What are adequate reconfiguration models?
- Which configurator approaches fit best for reconfiguration?

These questions can be summarized to the following key question:

> How can we systematically derive and document product transversal dependencies in practice so that reconfiguration can be largely automated by product reconfigurators?
> Here the term “in practice” means that simplifying suggestions such as “document all dependencies and the algorithm will work” are not considered to be useful. Instead, the engineer needs methods and approaches that prune the search space of configuration items to a manageable size.

Interestingly these reconfiguration challenges also exist in pure mechanical systems and are largely handled by the workshops. For example, an instrument panel bracket might be “enhanced” by an additional mounting hole, which is not used when the part is initially assembled. After some time, a new, additional part is attached at the mounting position.

When an older vehicle – lacking the mounting hole - comes to the workshop and is to be enhanced by new part, it is not possible to do this as no adequate mounting position exists. Problems like this are frequently solved in the shop, so what are differences to the EE world? Basically, drilling a new hole or even re-cabeling is on the re-engineering level of a well-trained mechanic or electrician, whereas ECUs and - even worse - software and parameters within ECUs are difficult or impossible to access.

Here, we can link back to the introduction: in recent years, an increasing amount of the customer benefit of vehicles has been implemented in electric and electronic components. In particular, software-based functions add an ever higher amount of customer value to our vehicles. Additionally creation and management of variability was improved by introducing parameters in ECUs, at least for production. But it turned after-sales a much more engineering-oriented than workshop-oriented work field.

From a reconfiguration perspective, this implies that the capability of reconfiguring complex software-based systems *systematically* will become more and more mandatory if we do not want to bind more and more development and after-sales capacities by reengineering complex systems.

4 Related Work

As far as we know, science has so far recognized reconfiguration more or less as an interesting supplementary question. For example, some configuration workshop papers of the AAAI, ECAJ, and IJCAI explicitly mention the problem. But also many papers don't even realize the challenge to change an existing system instead of configuring a new one.

One work that explicitly addresses the task is the doctoral thesis of Männistö (Männistö 2000). Here reconfiguration is modelled by reconfiguration operations. We could classify this as an "explicit upgrade package" approach in contrast to a "full reconfiguration model". Kreuz and Roller [Kreuz and Roller, 1999] describe the use of a reconfiguration system and explicitly address the problem of long term knowledge maintenance. The doctoral thesis of Heinisch [Heinisch, 2004] describes a configuration model for vehicles that allows for reprogramming ECUs.

Other publications do not aim at reconfiguration directly but possibly contribute to it: Is "retraction" in description logic based approaches (McGuinnes and Wright 1998) a useful full reconfiguration model? Does conceptual modeling (McGuinnes and Wright 1998) deliver a full reconfiguration model that is practically relevant? Can design-specifications and interactive configuration (Feldkamp, Heinrich, Meyer-Gramann, 1998) be also a solution for re-configuration in addition to re-use? How much and what kind of additional modelling is necessary to formally validate reconfigured configurations compared instead of validating only (Sinz, Kaiser Küchlein, 2003) production configurations? Can constraint solving heuristics (Hulubai, Freuder, Wallace, 2003) contribute to the problem of finding not only correct, but also optimal reconfigurations? The literature delivers many potential contributions to reconfiguration but no ready-to-use approach.

Yet the production oriented view seems to be engrained deeply in the mindset of all parties concerned: numerous configuration publications do not acknowledge the problem at all. The perception of engineers and manufacturing staff in practice, even those having years of experience with configuration, is not better. To make a long story short: up to now we find no theory, no methods, and no systems that can handle reconfiguration problems of a practical size.

Hence, the goal of this paper is not to provide solutions for reconfiguration but to re-introduce the importance of the topic from practical business cases in after-sales, to provide a well founded basis for discussion, to review existing work, and to motivate for future research.

5 Bibliography

(Faltings, B., Freuder, E. C. 1998): Special Issue on Configuration. IEEE Intelligent Systems 14(4) 1998 29-85

(Feldkamp, Heinrich, Meyer-Gramann, 1998) SyDeR - System design for reusability, Artifical Intelligence for Engineering Design (1998), Special issue on configuration, Cambridge University Press.

(Heinisch 2004) Konfigurationsmodell und Architektur für eine automatisierte Software-Aktualisierung von Steuergeräten im Automobil, PhD thesis. Eberhard-Karls-Universität zu Tübingen

(Hulubai, Freuder, Wallace, 2003) The Goldilocks problem, Artifical Intelligence for Engineering Design, Analysis and Manufacturing (2003), Special issue on configuration, Cambridge University Press.

(Kreuz and Roller, 1999) Kreutz, Ingo, and Dieter Roller. 1999. "Knowledge Growing Old in Reconfiguration Context". In Configuration Papers from the AAAI Workshop, AAAI Technical Report WS99, 05. AAAI Press. p. 54–58.

(Männistö, 2000) Männistö, Tomi. 2000. A conceptual modelling approach to product families and their evolution. PhD thesis. Helsinki University of Technology.

(Männistö, T. et. al., 1999) Framework and Conceptual Model for Reconfiguration. In Configuration Papers from the AAAI Workshop, pp. 59-64. AAAI Technical Report WS-99-05. AAAI Press, 1999.

(McGuinnes and Wright 1998) Conceptual modelling for configuration: A description logic-based approach, Artifical Intelligence for Engineering Design (1998), Special issue on configuration, Cambridge University Press.

(Sinz, Kaiser Küchlein, 2003) Formal methods for the validation of automotive product configuration data, Artifical Intelligence for Engineering Design, Analysis and Manufacturing (2003), Special issue on configuration, Cambridge University Press.

(Soininen, T., Stumptner, M., 2003) Introduction to Special Issue on Configuration. Artificial Intelligence for Engineering Design, Analysis and Manufacturing 17(1-2) (2003) 1-2

"Dealing" with Configurable Products in the SAP Business Suite

Albert Haag
SAP AG
Research & Breakthrough Innovation
Neurottstrasse 16, 69190 Walldorf, Germany
albert.haag@sap.com

Abstract

SAP provides a configurator engine as part of its business software offering already for over a decade. During this time integration issues have proven overall more challenging than the issue of configurator technology itself. The "variant configurator" inside the SAP R3 system (one of the leading integrated business software suites worldwide) is in active use by more than 1500 SAP customers. However, the growing importance of sales channels like the internet or field sales pose new challenges in that "end customers" are more directly involved in the configuration process. This leads to different expectations with regard to ease of use and cost of implementation.

1 Introduction

This paper summarizes selected experiences with product configuration functionality in SAP's business software suite from a technical, configurator oriented perspective. It is not meant as a guide to respective product offerings at SAP.

A detailed presentation of the functionality and modeling concepts at the heart of configuration in SAP is out of scope, but is partially given in [Haag, 1998]. General documentation on product configuration at SAP can be found at: http://help.sap.com[1].

1.1 Configurable Products[2]

Product business data typically consists of a product key (id) and a set of properties or attributes. For example, when ordering a box of crayons, a material number uniquely identifies the product itself. A textual and pictorial description is associated with the box for depiction in a catalog. The size and weight of the box (as well as lot sizes) is needed to process shipment.

For *configurable products* additional properties describe the individual item being sold. For example, a car being manufactured to a specific customer order will often contain a combination of features that are unique to that particular order. Although its serial number is a unique identifier, it is not efficient to pre-assign a product id to each feasible combination of features. In the sequel, the term *characteristic* refers to a property of a configurable product that describes individually configured items. (Elsewhere, the terms property, attribute, and characteristic are often used interchangeably).

A configurable product can be complex. It can have a very large number of characteristics and it can be composed of parts that are themselves configurable products (nested configuration).

1.2 Product Configurator

A product configurator is a software function that aids a business application in handling configurable products. The primary task of a product configurator is to check that a given value assignment for the characteristics of a product is consistent and complete in the context of the business application. In the case of nested configuration the consistency and completeness of the determined set of parts must also be verified. A further task is to guide an interactive user through the configuration process, i.e. inform them of the valid choices at each step.

Product configurators are often implemented in a project specific way, either from scratch or on top of an existing configurator engine. Like all software development such an endeavor often proves very costly to sustain.

1.3 Configuration in SAP Business Software

As a vendor of standard business software SAP provides integrated product configuration functionality as part of its business software offering, both inside and outside back-office applications.

Back-Office Integration

The SAP R3 system (an integrated suite of business applications) includes the *variant configurator*. Its main focus is on back-office applications (business documents being main-

[1] For the variant configurator look in tab "Documentation mySAP ERP" in area SAP ERP Central Component > Logistics > Logistics General (LO) > Variant Configuration (LO-VC). For the IPC (Internet Pricing and Configuration) look in tab "MySAP Business Suite in area SAP Customer Relationship Mgmt. > CRM Enterprise > Enterprise Sales > Quotation and Order Management > Sales Order Processing (Configure to Order)).

[2] The term *product* also refers to an immaterial offering such as a service.

tained in-house by employees of the business). The variant configurator is integrated into the relevant business applications, particularly *sales and distribution*.

Widespread use of the variant configurator began in 1994/1995. Currently, an estimated more than 1500 SAP customers are actively using it.

Channel Enabling

SAP also provides a Java-based configurator called IPC (Internet Pricing and Configuration), primarily to enable sales channels outside the back-office. The IPC not only addresses Internet sales channels (both B2B and B2C), but also call centers, field sales, etc.

Since orders are usually ultimately fed into a back-end (R3) system, compatibility with the variant configurator must be maintained as far as possible in standard application scenarios.

An "advanced mode" that sacrifices compatibility for more configurator functionality is available for custom projects where standard integration is not required. This is used in particular for configuration of complex systems and networks.

Productive use of the IPC began in 1998/1999. It is not yet used quite as pervasively as the variant configurator. However, potentially, every variant configurator customer is also interested in channel enabling.

2 Types of Configuration

Integration of product configuration in sales applications have shown that it is useful to distinguish the following three types of configuration:

- High-level configuration
- Manufacturing completion
- Low-level configuration

High-level configuration is primarily interactive configuration. A user constructs a consistent and complete configuration as part of a business document (e.g. sales order). The set of characteristics allows pricing, but is not necessarily extensive enough for direct derivation of manufacturing/ procurement information including detailed costing and availability.

Manufacturing completion is a non-interactive process. The configurator checks a pre-defined configuration (resulting from either high-level configuration or an external order-entry tool) for consistency and completeness. It also expands the configuration to a larger set of characteristics. This expansion is needed to drive the manufacturing processes, as well as determine availability of parts, costing, etc. It can depend on the production plant chosen.

Low-level configuration is non-interactive, programmatic process. The (expanded) characteristics of a single component drive the selection of parts in the manufacturing process and the configuration of routings etc.

The variant configurator and the IPC deal with high-level configuration and manufacturing completion. It has proven beneficial to provide a dedicated implementation for low-level configuration.

3 Selected Aspects of Integration

Most customers value end-to-end integration much more than enhanced configurator functionality. If R3 is the back-end business system, this means that any front-end configurator is best compatible with the variant configurator.

Externalized Configuration

The result of a (high-level) configuration must be stored with the business document (sales order). This configuration is used in several ways:

- Displayed/ printed as part of sales order
- Re-loaded by the configurator for further changes
- Read as input to low-level configuration
- May be modified directly on shop floor in extreme cases ("on the fly substitution of features")

Synchronization of Product Model with Master Data

Large parts of the product model used by the configurator must be in synch with master data used by various business processes. Particularly, high-level configuration and manufacturing completion must produce results that are intelligible to low-level configuration. Also, the space of legal configurations should match what can be (reasonably) manufactured.

Forecasting may benefit by being able to introspect the product model. For example, if the model dictates that all red cars have black seats, any forecast on read cars will also apply to black seats (simplistic case).

The variant configurator directly uses the R3 master data as its product model repository. The IPC automatically extracts product models from the R3 master data. However, many other configurators require product models expressed in their own modeling language. Keeping such product models in synch with the master data can be a costly exercise. Experience has shown that this cost is often prohibitive if the synchronization is manual.

Pricing, Costing, and Availability

Pricing should be possible solely based on the master data and parameters provided by the business document (such as customer and contract data). In contrast, costing and availability can entail a full simulation of the logistic processes. This requires all levels of the configuration actually to be carried out.

5 Managing the Cost of Implementing Product Configuration

Due to the many sources of additional complexity, dealing with configurable products can be costly compared to dealing with non-configurable products even when using an integrated environment such as provided by SAP. It is one major objective to reduce and better control this cost. Some factors that directly influence this cost:

- Complexity of the product and its model
- Power of the maintenance and test environment

- Implementation platform (performance, stability, and scalability)

5.1 Product and Product Model Complexity

Experience has shown that it is useful to distinguish at least the following three levels of product complexity:

- Simple products with a small number of characteristics (around 50)
- Nested products. Products that involve configuring components. Nesting can go several levels deep. Altogether there can be thousands of characteristics.
- Systems/ network configuration – there can be tens of thousands of items in such a solution each with many characteristics

The complexity of the product influences not only the performance of configuration and the complexity of the user interface, but also the complexity of integration with the business processes. In R3 there is fairly complete standard integration with sales and logistics processes for simple products and nested configurable products. Integration for systems configuration has been harder to standardize. This is usually done on a project basis using the "advanced mode" in the IPC.

Keeping the product model (including constraints) as simple as possible is a substantial part of controlling the cost of implementing configuration. Models for products that are essentially similar from the business point of view can differ widely in their complexity. Unwarranted complexity often ensues because the modeler is trying for perfection beyond the actual business need.

Currently good modeling is achieved by following "best practices" established by experienced modelers. An idea for further improvement is to remove sources of complexity by tailoring the modeling environment to suit the particular business need and/or product complexity. Examples of this at different levels might be:

- Allow simple products only (disallow components)
- Disallow multiple inheritance in classifying products
- Disallow constraints between components in a nested product, except for ancestor/ descendant relations.

5.2 Modeling and Test Environment

At a certain level of abstraction product modeling can be seen as software development. Hence, a modeling environment should resemble a modern software IDE (Integrated Development Environment). Some key features are:

- Allow overview and navigation within the model
- Allow immediate simulation of the model using the configurator
- Support for "best practices" as discussed above.

One idea for further improvement is to provide a direct analysis of performance and integration implications based on the complexity of the constraints and estimated size of ensuing configurations. Also, support for customizing the user interface is needed. However, this is usually done by persons distinct from the modeling, so it is best provided separately.

To support testing, the configurator should provide profiling, tracing and logging facilities. The IPC also foresees debugging facilities for the evaluation of constraints. However, this feature seems not to have been needed to date.

5.3 Implementation Platform

The variant configurator runs within the R3 system and thus benefits from the performance, stability, and scalability of the underlying SAP server technology as well as being able to use its central repository. The IPC being implemented in Java has hitherto not been able to make direct use of this platform. However, it is currently also being moved to the SAP technology stack. This move is expected to yield a substantial reduction in cost in installing, administering and running the IPC.

6 Recent Challenges and Current Work

Non-professional end-users require better support during interactive configuration. Mostly this means improvements to the user interface. To some extent the core configurator functionality must be extended. Some recent improvements to the core configurator include:

- Calculation of the set of valid values for a characteristic assuming the current value were not set
- Information about how to resolve conflicts and to make excluded choices available
- More rigorous handling of conflicting defaults

By chance, all of the above involve improving or making better use of the TMS (Truth Maintenance System) which is a part of the IPC. For a description of the role of defaults and of the TMS in the SAP configurators see [Haag, 1998] .

6.1 Calculating "Dynamic User Domains"

One very basic function of a configurator is to calculate the set of available choices at each step in an interactive configuration. If a value for a characteristic has been set (by the user or as a default) the user needs to know what values would be available for that characteristic to replace the current choice. This set of is called the *dynamic user domain* at SAP.

A basic means (used by some customers for some time) to address this is to perform the following steps in the background each time a dynamic user domain is to be displayed:

- Delete the current value and reconfigure
- Cache the resulting domain for the characteristic
- Reset the value and reconfigure. Display the cached domain as the set of available values to the user

Ideally this procedure needs to be applied only on demand for one characteristic at a time. However, in web scenarios all currently displayable characteristics need to be processed on each round trip (to avoid additional round

trips). Thus, the above approach works for simple products, but has proved to have performance problems for the other cases. The improvement is to emulate this procedure internally:

- Simulate retracting user and default justifications for the current value in the TMS. Do not actually delete anything from the internal configuration state
- Perform some additional constraint propagation as warranted (only a fraction of the effort needed to reconfigure after deletion of the value)
- Cache the resulting domain for the characteristic
- Reinstate any justifications that were retracted in the first step (no further constraint propagation required)

Experiences so far suggest that this performs also for nested products and systems configuration[3]. However, it cannot take into account the effect of values that were set by procedural means because justifications are incomplete in this case. In other words a purely constraint-based model is works best.

Note that the calculation described above could be generalized to *what-if* type queries. However, handling these would require conceptual extensions to the user interface.

6.2 Guided Resolution of Inconsistencies

For some time the TMS in the IPC has been extended to perform an ATMS label calculation[4] on demand. The TMS calculates minimal sets of user choices and/ or defaults that support inconsistency, called *nogoods*.

In order to resolve inconsistency one user input/ default needs to be retracted for each nogood. The user interface can guide the user to choose one user input/ default for retraction from a list. Each entry in the list will be in at least one nogood. Entries that are in multiple nogoods are presented more prominently, as this will handle multiple nogoods in one step. The procedure must be repeated until all nogoods have been handled.

The same procedure might also be used to allow selecting choices that have been excluded by the constraints. In this case the newly chosen value would not be offered for retraction. An explanation of what the choosing the excluded value entails could be obtained by calculating the ATMS label pertaining to the exclusion.

6.3 Handling of Defaults

During Configuration it can happen that inconsistency is derived and is founded on one or more values originally set as *defaults*. Currently there is an algorithm in place that bumps defaults in such situations. However, the exact effect of this bumping in large models is not transparent to the modelers and thus hard to manage and test. Also it has proven difficult to reinstate these defaults when the cause of their having been bumped is retracted.

For this reason there is work in progress to perform this bumping in a more rigorous manner and to reinstate bumped defaults when warranted based on their membership in nogoods.

[3] The dynamic user domain calculation can be turned off in the event performance problems are encountered

[4] For a conceptual overview of an ATMS (Assumption Based Truth Maintenance System) see [DeKleer, 1986]. The details of how the TMS in the IPC relates user choices and justifications to assumptions have not yet been published. Ideas put forth in [Haag, 1998] have since been carried further.

7 Acknowledgements

Many colleagues (both in development and consulting), customers, and third parties have all contributed to developments and improvements of product configuration at SAP for over a decade. The following website is maintained by a group representative of this effort:
http://www.configuration-workgroup.com/

References

[DeKleer, 1986] J. De Kleer. An assumption based truth maintenance system. Artificial Intelligence, 28:127--162, 1986

[Haag, 1998] Albert Haag, Sales Configuration in Business Processes. *IEEE Intelligent Systems* 13(4): 78-85 (1998).

Configurators in innovative or standardized business processes

Gerhard Fleischanderl
Siemens AG Österreich
Program and System Engineering
Erdberger Laende 26, A-1030 Vienna, Austria
gerhard.fleischanderl@siemens.com

Abstract

Business processes for sales and distribution were highly improved by the application of configurators. As more companies employ configurators, the sales process with configurators is not innovative any more, but becomes standardized. A company may again come up with a business process innovation if it launches radically innovative configurator features.

1 Innovation and standardization

In his book [Moore, 2002] G. Moore describes the life-cycle stages of services and business processes.

- Stage 1: Invention
- Stage 2: Innovation
- Stage 3: Standardization
- Stage 4: Commoditization

In the innovation stage, companies design a service in a way that yields competitive advantage through differentiation. When a service is successful, it is often copied by competitors and its basic idea is used widely. The transition from innovation to standardization also means that the key success factor changes from differentiation (i.e. features) to productivity (i.e. efficiency and price). A service moves from the standardization to the commoditization stage when many suppliers offer the same service.

2 Current situation of configurators

Ten years ago the business processes with configurators were in the innovation stage. Only few configurator tools were available, and few were employed in practical applications. Differentiation of business processes by the application of a configurator was important.

Configurators are now widely accepted. Their usefulness was proven by many productive applications. Now the business processes with configurators are in the standardization stage. The configurator tools may be innovative, but the sales processes are standardized (in a sense) as most sales processes employ configurators.

The components of a configurator application are relevant in different stages of the life-cycle of the sales process (D = distinguishing factor for companies that use a configurator).

Stage	*Invention*	*Innovation*	*Standardization*	*Commoditization*
Configurator kernel	D	D		
Requirements engineering		D	D	D
Interfaces (files, UI)			D	D

If radically innovative configurator features are developed and deployed, a company may again come up with an innovative business process and differentiate itself from the competitors. If the features of configurators broadly remain as they are, the business process with configurators will inevitably move from the standardization to the commoditization stage. There only cost counts.

3 Summary and conclusions

The features of the kernel itself become less important as configurator applications become established. The requirements engineering is a key success factor in all stages that are relevant for practical use. The interfaces become more important in the stages where productivity is the key driver for competitive advantage.

Altogether configurator applications are successful if they support the customers in being more competitive. Researchers and developers want to work on innovative features. However, the competitive advantage for the customers may require a focus on productivity, i.e. increased efficiency for the customer's processes.

References

[Moore, 2002] Geoffrey A. Moore. *Living on the Fault Line, Revised Edition: Managing for Shareholder Value in Any Economy*. HarperCollins Publishers, Inc., New York, 2002.

MCml2

Jiří Vyskočil, Martin Trčka, Libor Denner, Petr Šváb
Charles University in Prague, Faculty of Mathematics and Physics
Malostranské náměstí 2/25, 118 00 Praha 1, Czech Republic
jiri.vyskocil@mff.cuni.cz

Abstract

We decribe the MCml2 system which generates applications for configuration. Typical usage of this system is configuration of the Linux kernel, however it is not restricted to any platform, programming language, user interface or configuration task.

1 Introduction

1.1 Motivation

Imagine the following situation: you have just bought a new external CD-Writer HP 8230e and you want to use it under Linux operating system. First you probably need to add support for this drive to Linux kernel. But Linux kernel configuration is not easy, you must not forget to enable SCSI and USB support, input core support, ISA- and PCI-bus hardware support, USB mass storage support and so on, otherwise your CD-Writer will not work. Laic user usually has not sufficient knowledge and experience to find out, realize or even remember such dependencies.

To make user's work easier or even to make it possible to be done we need some expert system, which can, on the basis of knowledge acquired from Linux kernel specialist, solve these bindings itself or with minimal assistance of user.

In the situation above, the user would need to know only the type of CD-Writer and the expert system would set up all configuration items necessary for work with this writer.

1.2 What does MCml2 do?

MCml2 project generates applications, expert systems that allow setting a configuration. A configuration is a set of configuration symbols that have its own type and value, that have to conform to some rules and that have some dependencies.

MCml2 input is a file with declaration of symbols, rules and dependencies written in CML2+ language [2]. This file represents knowledge base and is usually called CML2+ rulebase.

MCml2 output is an application that "realizes" this input, i. e.

- allows setting of symbol values
- checks and, in case of need, forces the validity of configuration, in other words it watches over rules and dependencies
- automatic computes dependent values
- explains all configuration inconsistency to user

1.3 Features and Advantages of MCml2

Generally, MCml2 implements CML2+ language [2] , which is CML2 [3] with some extensions. CML2+ is a language for describing sets of variables (called symbols) and was developed to be a tool for linux kernel configuration. It's a descriptive language, not imperative. It means you only describe what should be satisfied and some CML2+ engine reads it and performs it (all details about CML2+ can be found in CML2+ specification [2]). MCml2 will be - in a way - such an engine. CML2+ is extended two value propositional logic. There are extensions such as arithmetic operations, tree state logic, strings operations, etc.

The MCml2 system is not actually the final desired program that works with CML2+ description, but a generator of all such programs. And by 'all' we mean in all languages and all user-interfaces that we support now or in the future.

Here are some basic points concerning features and advantages of the MCml2 system:

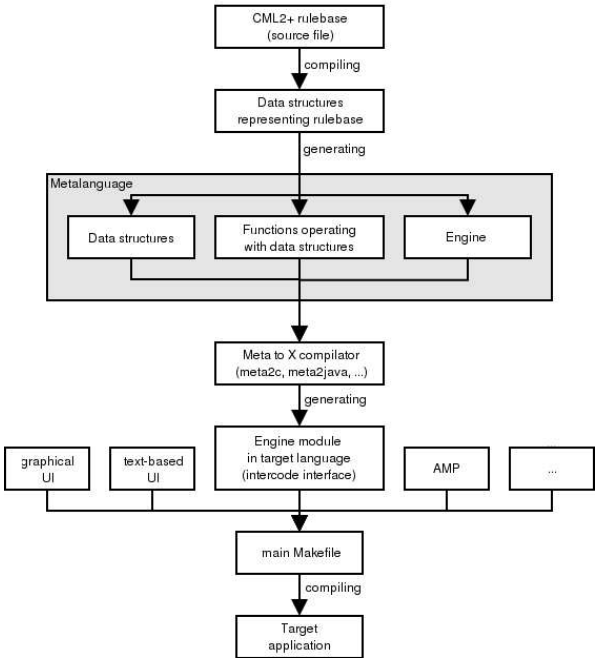
- The MCml2 generator is written purely in C, so it could be compiled on wide range of architectures and operating systems.
- MCml2 uses own extension of CML2 language that better fits needs of configuration systems than older CML1. It also supports UTF-8, so texts of configuration symbols are not limited to English only.
- MCml2 is able to generate code in several procedural programming languages (at this time MCml2 supports C and Java), so there is larger variety of programmers that can extend final applications.
- Code generated by MCml2 is fast because it is optimized just for one purpose.
- Presentation part of final applications is strictly separated from the computing part; therefore it is not complicated to change the interface of applications.
- Various updates in wide range of future extensions are easy to achieve.

The whole concept of MCml2 development has an idea of universality. We don't want CML2+ to be restricted only to

C-code and linux configuration. That's why the result application needn't be only in C, but in any other procedural language (which is or will be supported). One of them is Java for ensuring multiplatform usage and for possibility of web-configuring without downloading any application.

2 Project Structure

The basic project structure is clearly shown in the Project dataflow graph.



Here you can see how the source code becomes the result program of the final user. One can recognize four major steps:

- CML2+ compilation
- Metacode generation
- Meta to X compilation
- Linking with the desired User Interface

2.1 CML2+ Compiler

The raw text CML2+ data are compiled by CML2+ compiler. It is called from the generator and performs all lexical, syntax and semantic analysis. It also generates tags to help texts and expression strings and other data that are used by further and final parts.

For CML2+ debugging purposes it's possible to make just a very fast CML2+ checker - the standalone compiler, which is quite useful if you just want to check whether your rulebase change is OK.

2.2 Metacode Generator

The data from semantic analysis are transformed to metalanguage code. Metalanguage code [4] is high level Pascal-like procedural language.

This process runs separately, CML2+ formulae and data structures are dynamically generated and the basic engine is mainly added as a static metalanguage code.

This is all performed with no Metalanguage in text form. It's because all the Metalanguege stuff is passed in memory, and therefore it never occurs in non-parsed form.

2.3 Meta to X Compilation

The Metalanguage must be compiled into the target language to be usable. That's why the third part is the Meta to target-language compiler. For example only the Meta to C compilation is typically used for Unix kernel rulebase. Meta to Java would be probably used mainly for web purposes.

This is a part of generator binary and it is called at the end according to the command-line options. After this step the whole engine is generated in the target language.

2.4 Linking with the desired User Interface

In the final step comes the global makefile which links together the generated engine and the prepared user interface. It could be one of the supported user interface or any other that communicates properly through intercode. It, of course, has to be in an appropriate language.

After this final link the standalone end-user application is built and configuration may begin.

3 Applications and Similar Projects

The original idea of universal configuration tool is based on Eric Raymond's Linux configuration language CML2. It's the ancestral language of our CML2+ (which is CML2 with extensions). Gcml2 project is another challenge, but it again solves only Linux kernel configuration problem. Comparison of MCml2 and other projects is not simple because other projects have different orientation.

The presented system was tested on some versions of the Linux kernel and some other applications. It can be downloaded from sourceforge site [1]. Many tests show that this configuration tool in Linux kernel is faster for the end user than any other existing configuration tool. Due to its versatility it can be used for numerous other real world configuration tasks.

References

[1] Homepage of project MCml2
`http://cml2config.sourceforge.net/`

[2] CML2+ language specification
`http://cml2config.sourceforge.net/doc/doc-user-cml2plus.html`

[3] Original CML2 language specification by Eric S. Raymond
`http://www.catb.org/~esr/cml2/cml2-reference.html`

[4] Meta language specification
`http://cml2config.sourceforge.net/doc/doc-prog-meta.html`

Tacton Configurator - Research Directions

Klas Orsvarn, Tacton
email: klas.orsvarn@tacton.com

Tacton Configurator is based on 18 years of research and application in product configuration and knowledge based systems, starting at the Swedish Institute of Computer Science in 1987. The four key capabilities that we focus on are the following:

1. empowering end-users to configure optimal configuration solutions based on customer needs
2. enabling product managers and engineers to build and maintain product specific configuration logic without programming
3. seamless integration with related business applications, such as CRM, e-commerce, ERP, PLM, and CAD
4. ability to solve any real-life configuration problem, meeting the above three criteria

We have very wide experience of successfully applying Tacton Configurator in North America and Europe in all kinds of equipment manufacturing industries, from large scale complex electronics, through mechanical engineering in assemble-to-order and make-to-order, to life sciences equipment, as well as services and project configuration.

Nevertheless, the configuration task is NP-complete, so there is always room for more research into even better solutions.

1. Arc consistency propagation is not enough to meet capability #1, and Tacton Configurator employs complete propagation as well as optimization search.
2. For capability #2, debugging of constraint systems is an immature research area.
3. Capability #3 requires real-time application of product and customer data from other applications.
4. Capability #4 requires highly dynamic CSP that is missing in many solutions, and analysis of different types of real-life configuration problems is an area in need of further research.