FAST BACKTRACK-FREE PRODUCT CONFIGURATION USING A PRECOMPILED SOLUTION SPACE REPRESENTATION

Tarik Hadzic, Sathiamoorthy Subbarayan, Rune M. Jensen, Henrik R. Andersen, Jesper Møller*, and Henrik Hulgaard*

Department of Innovation
IT University of Copenhagen
Glentevej 67
2400 Copenhagen NV
Denmark
+45 3816 8888
Email: {tarik, sathi, rmj, hra}@itu.dk

Configit Software A/S*
Vermundsgade 38 B
DK-2100 Copenhagen Ø
Denmark
+45 7022 6700
Email: {jm, henrik}@configit-software.com

In this paper we describe a two-phase approach to interactive product configuration. In the first phase, a compressed symbolic representation of the set of valid configurations (the *solution space*) is compiled offline. In the second phase, this representation is embedded in an online configurator and utilized for fast, complete, and backtrack-free interactive product configuration. The main advantage of our approach compared to online search-based approaches is that we avoid searching for valid solutions in each iteration of the interactive configuration process. The computationally hard part of the problem is fully solved in the offline phase given that the produced symbolic representation is small. The employed symbolic representation is Binary Decision Diagrams (BDDs). More than a decade of research in formal verification has shown that BDDs often compactly encode formal models of systems encountered in practice. To our experience this is also the case for product models. Often the compiled BDD is small enough to be embedded directly in hardware. Our research has led to the establishment of a spin-off company called Configit Software A/S. Configit has developed software for writing product models in a strongly typed language and has patented a particularly efficient symbolic representation called Virtual Tables.

Significance: Several companies have benefited from the tools developed by Configit Software. The application areas are diverse and include ordinary product configuration as well as sales support and user interfaces for hardware components.

Keywords: Interactive Product Configuration, Configuration Space Models, Binary Decision Diagrams.

1. INTRODUCTION

The focus in manufacturing industry has shifted from mass production to mass customization. Companies continually have to offer more product variants with greater flexibility. At the same time, the rapid development of information technology has significantly increased the complexity of each individual product. These changes have led to a situation where the industry is losing track of the functionality of their products. It has become common practice to ship mobile phones and software that is close to impossible to setup correctly even for expert customers.

Efficient tools are needed to handle the increasing complexity of products. Product configurators are one such class of tools. Given a set of rules defining the set of valid configurations (the *solution space*) of the product, the configurators guide sales people and users to find a valid and desirable configuration of the product. Most configurators are based on searching online in the solution space [8, 9, 14]. This may work well for many products, but it is impossible to guarantee that the search time is polynomially bounded with the size of the product model, since finding just a single valid configuration is NP-complete. This means that search-based configurators sometimes may have undesirable long response times. Moreover, the performance of these tools often depends on how the rules are written, and it can be very difficult to write rules that work well in practice.

In this paper we describe an alternative approach to product configuration based on a precompiled representation of the solution space. The approach has two phases. The first phase is offline and consists of compiling the product rules into a Binary Decision Diagram (BDD) [2] representing the solution space. BDDs are a canonical representation of Boolean functions. During the last 15 years, they have been applied successfully in formal verification and other areas of computer science to represent formal models of very large systems [3]. Our experience is that BDDs also compactly encode the solution space of industrial products. Since the compiled BDD is canonical, it only depends on what Boolean function the

product rules represent and not on how the rules are written. This gives the rule writer freedom to choose a format of the rules that naturally represents the behavior of the product.

In the second phase, the compiled BDD is used online in an interactive configurator. In each iteration of the interactive configuration process, specialized BDD algorithms compute the set of possible ways the current partial configuration can be extended to a valid product. The interactive configuration process is complete and backtrack-free. The user can choose freely between any valid configuration and is prevented from reaching dead-ends of impossible configurations. More importantly, the worst-case response time only grows polynomially with the size of the BDD. Thus, the computationally hard part of the configuration problem is fully solved in the offline phase given that the compiled BDD is small. Surprisingly this is often the case even for complex products with long compilation times.

Our research has led to the establishment of a spin-off company called Configit Software A/S. Configit has improved the BDD-based technique and patented a particularly efficient symbolic representation called Virtual Tables (VTs). A VT is an XML file that in addition to the symbolic representation holds the definition of the information it stores. VTs can be embedded in a wide variety of products ranging from web-configurators to electronic products.

The remainder of the paper is organized as follows. In Section 2, we formally define product configuration and describe the interactive configuration process. In Section 3, we show how to encode a solution space symbolically as a Boolean function and illustrate how this can be done with BDDs. In Section 4, we briefly introduce Configit's approach and special features. Section 5 presents experimental work. Related work is discussed in Section 6. Finally in Section 7, we conclude and consider directions for future work.

2. INTERACTIVE PRODUCT CONFIGURATION

We can think of product configuration as a process of specifying a product defined by a set of attributes, where attribute values can be combined only in predefined ways. Our formal definition captures this as a mathematical object with three elements: variables, domains for the variables defining the combinatorial space of possible assignments and formulae defining which combinations are valid assignments. Each variable represents a product attribute, variable domain refers to the options available for its attribute and formulae specify the rules that the product must satisfy.

Definition 1. A *configuration problem* C is a triple (X, D, F), where X is a set of variables $x_1, x_2, \ldots, x_n$, D is a Cartesian product of their finite domains $D_1 \times D_2 \times \ldots \times D_n$ and $F = \{f_1, f_2, \ldots, f_m\}$ is a set of propositional formulas over atomic propositions $x_i = v$, where $v \in D_i$, specifying conditions that the variable assignments have to satisfy.

Each formula f_i is a propositional expression φ inductively defined by

$$\varphi \equiv x_i = v \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \neg\varphi,$$

where $v \in D_i$. We will use the abbreviation $\varphi \Rightarrow \psi \equiv \neg\varphi \vee \psi$ for logical implication. For a configuration problem C, we define the *solution space* $S(C)$ as the set of all *valid configurations*, i.e. the set of all assignments to the variables X that satisfy the rules F. Many interesting questions about configuration problems are hard to answer. Just determining whether the solution space is empty is NP-complete, since we can reduce the Boolean satisfiability problem to it in polynomial time [10].

As an example consider specifying a T-shirt by choosing the color (black, white, red or blue), the size (small, medium or large) and the print ("Men In Black" - MIB or "Save The Whales" – STW). There are two rules that we have to observe: if we choose the MIB print then the color black has to be chosen as well, and if we choose the small size then the STW print (including a big picture of a whale) cannot be selected as the large whale does not fit on the small shirt.

The configuration problem (X, D, F) of the T-shirt example consists of variables $X = \{x_1, x_2, x_3\}$ representing color, size and print. Variable domains are $D_1 = \{black, white, red, blue\}$, $D_2 = \{small, medium, large\}$ and $D_3 = \{MIB, STW\}$. The two rules translate to $F = \{f_1, f_2\}$ where f_1 is $(x_3 = MIB) \Rightarrow (x_1 = black)$ and f_2 is $(x_3 = STW) \Rightarrow (x_2 \neq small)$. There are $|D_1 \| D_2 \| D_3| = 24$ possible assignments. Eleven of these assignments are valid configurations and they form the solution space shown in Figure 1.

(*black, small, MIB*)	(*black, large, STW*)	(*red, large, STW*)
(*black, medium, MIB*)	(*white, medium, STW*)	(*blue, medium, STW*)
(*black, medium, STW*)	(*white, large, STW*)	(*blue, large, STW*)
(*black, large, MIB*)	(*red, medium, STW*)	

Figure 1. Solution space for the T-shirt example.

When we talk about *interactive configuration*, we are referring to the process of a user interactively tailoring a product to his specific needs by using supporting software called a *configurator*. Every time the user assigns a value to a variable, the configurator restricts the solution space by removing *all* assignments that violate this new condition, reducing the available

user choices to only those values that appear in *at least* one configuration in the restricted solution space. The user keeps selecting variable values until only one configuration is left. The algorithm in Figure 2 illustrates this interactive process.

```
INTERACTIVE-CONFIGURATION(C)
1  Sol ← S(C)
2  while |Sol| > 1
3      do choose (x_i = v) ∈ VALID-ASS(Sol)
4          Sol ← Sol ∩ D_{x_i=v}
```

Figure 2. Interactive configuration procedure.

The VALID-ASS(*Sol*) procedure in line 3 extracts the set of valid assignments (choices) from the solution space *Sol*. We restrict the solution space in line 4 by intersection with $D_{x_i=v} = D_1 \times \ldots D_{i-1} \times \{v\} \times D_{i+1} \times \ldots D_n$, which effectively enforces that only those tuples with value *v* for x_i remain in the solution space.

This behavior of the configurator enforces a very important property of interactive configuration called *completeness of inference*. The user cannot pick a value that is not a part of a valid solution, and furthermore, a user is able to pick *all* values that are part of at least one valid solution. These two properties are often not satisfied in existing configurators, either exposing the user to backtracking or making some valid choices unavailable.

In the T-shirt example, the assignment $x_2 = small$ will, by the second rule, imply $x_3 \neq STW$ and since there is only one possibility left for variable x_3, it follows that $x_3 = MIB$. The first rule then implies $x_1 = black$. Unexpectedly, we have completely specified a T-shirt by just one assignment. Actually, the configurator just deletes all configurations that do not satisfy $x_2 = small$ and discovers that a solution space is reduced to just one tuple: (*black, small, MIB*).

From the user's point of view, the configurator responds to the assignment by calculating valid choices for undecided variables. It is important that the response time is very short, offering the user truly interactive experience. The demand for short response-time and completeness of inference is difficult to satisfy due to the hardness of the configuration problem.

3. TWO PHASE APPROACH

Since checking whether the solution space is empty is NP-complete, it is unlikely that we can construct a configurator that takes a configuration problem *C* and guarantees a response time that is polynomially bounded with respect to the size of *C*.

Our approach is offline to compile the solution space of a configuration problem to a representation that supports fast interaction algorithms. The idea is to remove the hard part of the problem in the offline phase. This will happen if the compiled representation is small. We cannot always avoid exponentially large representations. However, for most real-world problem instances, we get small representations and therefore fast interaction algorithms. Furthermore, after the compilation is finished, we know the size of the solution space representation. Therefore we are able to precisely predict the running time of the interaction algorithms.

3.1 Symbolic Solution Space Representation

Our configuration problem *C* can be efficiently encoded using Boolean variables and Boolean functions. We assume that domains D_i contain successive integers starting from 0. For example, we encode $D_2 = \{small, medium, large\}$ as $D_2 = \{0,1,2\}$. Let $l_i = \lceil \lg |D_i| \rceil$ denote the number of bits required to encode a value in domain D_i. Every value $v \in D_i$ can be represented in a binary format and therefore seen as a vector of Boolean values $\vec{v} = (v_{l_i-1}, \ldots, v_1, v_0) \in \mathbf{B}^{l_i}$. Analogously, every variable x_i can be encoded by a vector of Boolean variables $\vec{b} = (b_{l_i-1}, \ldots, b_1, b_0)$. Now, the formula $x_i = v$ can be represented as a Boolean function given by the expression $\vec{b} = \vec{v}$ i.e. $b_{l_i-1} = v_{l_i-1} \wedge \ldots \wedge b_1 = v_1 \wedge b_0 = v_0$.

In the T-shirt example, $D_2 = \{small, medium, large\}$ and $l_2 = \lceil \lg 3 \rceil = 2$, so we can encode $small \in D_2$ as 00 $(b_1 = 0, b_0 = 0)$, *medium* as 01 $(b_1 = 0, b_0 = 1)$ and *large* as 10 $(b_1 = 1, b_0 = 0)$. This translation to a Boolean domain is not surjective, i.e. not every combination of assignments to Boolean variables $b_{l_i-1}, \ldots, b_1, b_0$ yields a valid value $v \in D_i$. For example, the combination 11 does not encode a valid value in D_2. Therefore we introduce a Boolean constraint (a so called *domain constraint)* that forbids these unwanted combinations $F_D = \wedge_{i=1}^{n} (\vee_{v \in D_i} x_i = v)$. Furthermore, we define a translation function τ that maps a propositional expression φ to the Boolean function it represents

$$\tau(\varphi) : \prod_{i=1}^{n} \mathbf{B}^{l_i} \rightarrow \mathbf{B}.$$

The translation is defined inductively as follows

$$
\begin{aligned}
t(x_i = v) &\equiv (\vec{b}_i = \vec{v}) \\
t(\varphi \wedge ?) &\equiv t(\varphi) \wedge t(?) \\
t(\varphi \vee ?) &\equiv t(\varphi) \vee t(?) \\
t(\neg\varphi) &\equiv \neg t(\varphi).
\end{aligned}
$$

Finally, we are able to express a Boolean function representation $S'(C)$ of the solution space $S(C)$

$$S'(C) \equiv \bigwedge_{i=1}^{m} \tau(f_i) \wedge \tau(F_D).$$

The interactive process of product configuration can be represented using the already described procedure (Figure 2), but now using the Boolean representation of the solution space. The resulting algorithm is shown in Figure 3.

```
INTERACTIVE-CONFIGURATION(C)
1  Sol ← S'(C)
2  while |Sol| > 1
3      do choose (x_i = v) ∈ VALID-ASS(Sol)
4          Sol ← Sol ∧ τ(x_i = v)
```

Figure 3. Boolean version of interactive configuration.

3.2 Binary Decision Diagrams

A reduced ordered Binary Decision Diagram (BDD) is a rooted directed acyclic graph representing a Boolean function on a set of linearly ordered Boolean variables. It has one or two terminal nodes labeled 1 or 0 and a set of variable nodes. Each variable node is associated with a Boolean variable and has two outgoing edges *low* and *high*. Given an assignment of the variables, the value of the Boolean function is determined by a path starting at the root node and recursively following the high edge, if the associated variable is true, and the low edge, if the associated variable is false. The function value is *true*, if the label of the reached terminal node is 1; otherwise it is *false*. The graph is ordered such that all paths respect the ordering of the variables. A BDD representing the function $f(x_1, x_2) = x_1 \vee \neg x_1 \wedge \neg x_2$ is shown Figure 4a.

A BDD is reduced such that no two distinct nodes u and v are associated with the same variable and low and high successors (Figure 4b), and no variable node u has identical low and high successors (Figure 4c).

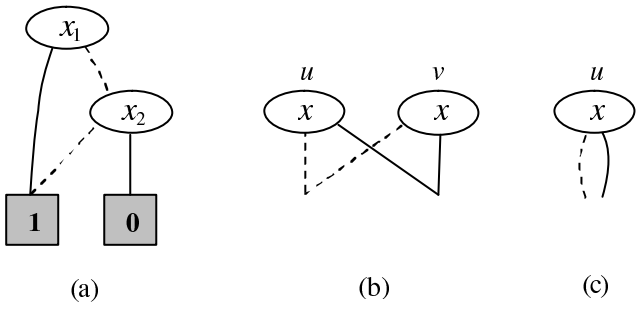

Figure 4. (a) A BDD representing the function $f(x_1, x_2) = x_1 \vee \neg x_1 \wedge \neg x_2$. High and low edges are drawn with solid and dashed lines, respectively. (b) Nodes associated with the same variable with equal low and high successors will be converted to a single node. (c) Nodes causing redundant tests on a variable are eliminated.

Due to these reductions, the number of nodes in a BDD for many functions encountered in practice is often much smaller than the number of truth assignments of the function. Another advantage is that the reductions make BDDs canonical [2]. Large space savings can be obtained by representing a collection of BDDs in a single multi-rooted graph where the sub-graphs of the BDDs are shared. Due to the canonicity, two BDDs are identical if and only if they have the same root. Consequently, when using this representation, equivalence checking between two BDDs can be done in constant time. In

addition, BDDs are easy to manipulate. Any Boolean operation on two BDDs can be carried out in time proportional to the product of their size.

The size of a BDD can depend critically on the variable ordering. To find an optimal ordering is a co-NP-complete problem in itself [2], but a good heuristic for choosing an ordering is to locate dependent variables close to each other in the ordering. For a comprehensive introduction to BDDs and *branching programs* in general, we refer the reader to Bryant's original paper [2] and the books [13, 18].

3.3 BDD-Based Interactive Configuration

In the offline phase of BDD-based interactive configuration, we compile a BDD $\tilde{S}(C)$ of the Boolean function representation $S'(C)$ of the solution space. The variable ordering of $\tilde{S}(C)$ is identical to the ordering of the Boolean variables of $S'(C)$. $\tilde{S}(C)$ can be compiled using a BDD version $\tilde{\tau}$ of the function τ, where each Boolean operation is translated to its corresponding BDD operation

$$\begin{aligned}
\tilde{\tau}(x_i = v) &\equiv \text{BDD of } \tau(x_i = v) \\
\tilde{\tau}(\varphi \wedge \psi) &\equiv \mathrm{Op}_{\wedge}(\tilde{\tau}(\varphi), \tilde{\tau}(\psi)) \\
\tilde{\tau}(\varphi \vee \psi) &\equiv \mathrm{Op}_{\vee}(\tilde{\tau}(\varphi), \tilde{\tau}(\psi)) \\
\tilde{\tau}(\neg\varphi) &\equiv \mathrm{Op}_{\neg}(\tilde{\tau}(\varphi)).
\end{aligned}$$

In the base case, $\tilde{\tau}(x_i = v)$ denotes a BDD of the Boolean function $\tau(x_i = v)$ as defined in Section 3.1. For each of the inductive cases, we first compile a BDD for each sub-expression and then perform the BDD operation corresponding to the Boolean operation on the sub-expressions. We have

$$\tilde{S}(C) \equiv \mathrm{Op}_{\wedge}(\tilde{\tau}(F_D), \tilde{\tau}(f_1), \cdots, \tilde{\tau}(f_m)).$$

Due to the polynomial complexity of BDD-operations, the complexity of computing $\tilde{S}(C)$ may be exponential in the size of C.

A version of the INTERACTIVE-CONFIGURATION procedure shown in Figure 3, where the Boolean functions are represented by BDDs, is used in the online phase. Each Boolean operation of this procedure is translated to its corresponding BDD operation. The response time of the procedure is determined by the complexity of performing a single iteration of the procedure. All sub-operations can be done in time linear in the size of *Sol* except VALID-ASS in Line 3. This procedure can be realized by a specialized BDD operation with worst-case complexity

$$O\left(\sum_{i=1}^{n} |V_i| |D_i|\right),$$

where V_i denotes the nodes in *Sol* associated with BDD variables encoding the domain of variable x_i. As usual, D_i denotes the domain of x_i. For each value of each variable the procedure tracks whether the value is encoded by *Sol*. Due to the ordering of the BDD variables, for each variable x_i, this tracking can be constrained to the nodes V_i.

4. CONFIGIT SOFTWARE

Configit Product Modeller (Configit-PM) is a software product for interactive configuration developed and distributed by Configit software [4]. Configit-PM requires product models to be specified in its own strongly typed language. The language is simple but expressive enough to handle the product models occurring in real-life modeling applications. Configit-PM also has a compiler which first checks for the semantic correctness of the product model. If the semantics is valid then it creates a virtual table (VT) that contains all valid solutions of the product model. VTs are stored in XML format. Details about the variables in the product model including their domain sizes are stored in a header part of the VT file. The header is followed by a BDD derived data structure that represents the valid solutions of the product model. All the information necessary to configure a product is embedded in a single VT file.

Using Configit-PM, a preferred interface can be created to interact with the VT file for configuring a product. The Configit-PM also has a built-in simulator, PM-Viewer, which can be used to interact and configure products. PM-Viewer can be used to select a value for each variable in the product model. It also has an undo facility to go back and forth between selections. Implications of the user selections will be shown as forced selections. The user need not have to select values for all the variables. After selecting a value for zero or more variables, the user can request the PM-Viewer to complete the rest of the options automatically. The PM-Viewer can also give explanations for all the forced selections. When the user wishes to select an invalidated value of a variable, the PM-Viewer will show a list of previous choices to be undone to resolve the conflict.

5. EXPERIMENTAL WORK

In this section we present experiments we have carried out using Configit-PM. The results are listed in Table 1. First column lists a benchmark name. Four benchmarks were used in the experiments. Three successive columns list the time taken for generating the corresponding VT, the size of the VT, and, the number of valid configurations (#Solutions), respectively. Last column shows the average response time over 1000 random requests on the generated VT. Each request corresponds to an iteration of the INTERACTIVE-CONFIGURATION procedure. The Renault benchmark represents a car configuration problem used in [1]. The PSR benchmark represents a power supply restoration (PSR) problem. Information about the PSR problem is available in [16]. PC is a benchmark distributed by Configit [4] along with Configit-PM. It represents a personal computer configuration problem. Parity represents a parity learning problem as a configuration instance. Information about this problem is available in [7]. The results show that even though a problem may take long time to compile in the offline phase, it may result in a small VT that gives very short response times in the online phase. This is in particular true for the Renault benchmark. Also notice that VTs due to the symbolic encoding may represent large solution spaces very compactly.

Benchmark	Virtual Table			Average Response Time (sec)
	Time (sec)	Size (KB)	#Solutions	
Renault	460.00	1292	2.8×10^{12}	0.127
PSR	0.38	37	7.7×10^{9}	0.001
PC	0.89	24	1.1×10^{6}	0.075
Parity	30.00	1219	198×10^{6}	0.096

Table 1. Experimental results on four benchmark problems .

6. RELATED WORK

Related work can broadly be classified into search-methods based on Constraint Satisfaction Problems (CSPs) [9] and Boolean satisfiability formulations [14], and, compilation methods using data structures like acyclic constraint networks [5, 6] and Automata [1]. In this section we give an overview of some of them.

In [8], the authors propose a preprocessing method to convert a CSP into another one, having Backtrack-Free problem Representation (BFR). The BFR can then be used for interactive configuration. Unlike conventional preprocessing methods which add additional constraints and hence increase the size of the problem representation, their method does not add any additional constraints. Instead, they restrict the domain of variables, such that a BFR is obtained. The main drawback of this approach is that the BFR does not contain all valid solutions. They give a guarantee that their preprocessor will give an error message if all solutions are removed by it. Although the authors claim that deletion of some valid solutions by their preprocessing step is acceptable in many cases, hiding even a single valid solution from the user is questionable. Although their representation is backtrack-free like our BDD-based method, some solutions are lost and hence their method is not preferable in real-life product configuration systems. In [9], the authors use consistency methods to obtain *explanations* and *implications* for a configurator based on a CSP representation. They use the N-Queens problem to demonstrate their method. It is like other CSP-based search methods, which solve an intractable problem every time the user makes a request. In [18], the authors presented Minimal Synthesis Trees (MSTs), a data structure to compactly represent the set of all solutions in a CSP. It takes advantage of combining the consistency techniques with a decomposition and interchangeability idea. Unlike our approach, which generates worst case exponential-size BDDs, the MST is a polynomial-size structure. Operations on the MSTs, however, are of exponential time complexity while they are of polynomial complexity in our approach.

In [14], the authors present an approach to the configuration problem based on a Boolean Satisfiability (SAT) solver. They have developed a non-interactive configuration system (BIS) based on a new SAT solver designed by them. The BIS system was developed for a commercial car manufacturer. Although their technique can be extended to an interactive configuration method, it will not move the intractability of the configuration process into an offline activity.

The problem with search-based methods is that the intractability of the configuration problem is solved every time the user gives a request to the configurator. In case of compilation techniques, the advantage is that the compilation process may be intractable but once the valid solutions are compiled into an efficient data structure, the interaction process is efficient.

Acyclic constraint networks and the *Tree clustering algorithm* [5, 6] represent a CSP solution space in a more compact way, organizing it as a tree of solved sub-problems. The generated structure offers polynomial time guarantees for extracting a solution in the size of the generated structure. The size of the sub-problems, however, cannot be controlled for all instances and might lead to an exponential blow-up. The complexity of the original problem is dominated by the complexity of the sub-problems, which are exponential in both space and time. Nevertheless, this is one of the first compilation approaches used to solve CSP problems. There are efforts to cope with this exponential blow-up by additional compression using Cartesian product representation [12].

In [1], the authors present a method which compiles all valid solutions of a configuration problem into an automaton. After compiling the solutions into an automaton, functions required for interactive configuration, like implications, explanations, and valid-domain-calculations can be done efficiently. They also present a theoretical view of all the complexity issues involved in their approach. They show that all the tasks involved in an interactive configuration process are intractable in the worst case. They claim that intractability can be circumvented by compiling configuration problems into an automaton. That is, moving the intractable part of the problem into an offline compilation process. Technically this work is the closest one to our approach. They use automaton to represent valid configurations, where we use BDDs. The two approaches to two-phase interactive configuration may perform equally well. However, a major advantage of using BDDs is that this data structure has been studied intensely in formal verification for representing formal models of large systems [3, 19]. In particular, the variable ordering problem is well studied [13]. Furthermore a range of powerful software packages have been developed for manipulating BDDs [11, 15]. To our knowledge, the automata approach has not reached this level of maturity.

7. CONCLUSION

In this paper we have demonstrated how to solve the computationally hard interactive configuration problem by dividing it into two phases. First, in an offline phase, we compile the solution space of the problem to a Boolean domain using BDDs as underlying data structure. Second, if the resulting BDD is small enough, we achieve fast algorithms for interactive configuration in an online phase while providing user-friendly requirements such as completeness of inference.

The experimental results indicate that BDDs representing the solution space of real-world configuration problems are often small and enable very short response times in the online interactive configuration phase. Future work includes combining our approach with the state-of-the-art search-based techniques for solving the configuration problem. We also plan to generate more experimental results using benchmarks from different application domains.

REFERENCES

1. J. Amilhastre, H. Fargier, P.Marquis: Consistency restoration and explanations in dynamic CSPs—application to configuration. Artificial Intelligence, 135(1-2), 199-234, 2002

2. R. Bryant: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers. Vol. C-35, No. 8, 677-691, 1986.

3. J. R. Burch, E. M. Clarke, K. McMillan: Symbolic model checking: 10^{20} states and beyond. In Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science, pages 428–439, 1990.

4. Configit website : www.configit-software.com

5. R. Dechter, J. Pearl: Network-based heuristics for constraint-satisfaction problems. Artificial Intelligence. Vol. 34 No. 1, 1-38, 1987

6. R. Dechter, J. Pearl: Tree-Clustering Schemes for Constraint-Processing. Artificial Intelligence. Vol. 38(3), 353-366, 1989

7. ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/contributed/crawford/README

8. E.C.Freuder, T.Carchrae, J.C.Beck: Satisfaction Guaranteed in Workshop on Configuration, Eighteenth International Joint Conference on Artificial Intelligence, 2003

9. E. C. Freuder, C. Likitvivatanavong, R. J. Wallace: Explanation and Implication for Configuration Problems in IJCAI-2001 Workshop on Configuration, Seattle, WA, August 2001, 2001

10. M.R. Garey, D.S. Johnson: Computers and Intractability-A Guide to the Theory of NP-Completeness, W H Freeman & Co, 1979

11. J. Lind-Nielsen: BuDDy - A Binary Decision Diagram Package. Technical Report IT-TR: 1999-028, Institute of Information Technology, Technical University of Denmark, 1999. `http://cs.it.dtu.dk/buddy`.

12. J. N. Madsen: Methods for Interactive Constraint Satisfaction. M.Sc. Thesis, Department of Computer Science, University of Copenhagen. 2003. http://www.diku.dk/forskning/performance-engineering/jeppe/

13. C. Meinel, T. Theobald: Algorithms and Data Structures in VLSI Design. Springer, 1998

14. C. Sinz, A. Kaiser, W. Küchlin: Formal methods for the validation of automotive product configuration data. Artificial Intelligence for Engineering Design, Analysis and Manufacturing, 17(1):75-97, January 2003. Special issue on configuration.

15. F. Somenzi: CUDD: Colorado university decision diagram package. `ftp://vlsi.colorado.edu/pub/`, 1996.

16. S. Thiebaux, M. O. Cordie: Supply restoration in power distribution systems a benchmark for planning under uncertainty. In Pre-Proceedings of the 6th European Conference on Planning (ECP-01), 85-96, 2001

17. I.Wegener: Branching Programs and Binary Decision Diagrams. Society for Industrial and Applied Mathematics (SIAM), 2000

18. R. Weigel, B. Faltings: Compiling constraint satisfaction problems. Artificial Intelligence 115(2): 257-287 (1999)

19. B. Yang, et.al,:A Performance Study of BDD-Based Model Checking. Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design, 255 - 289, 1998