Comparing Two Implementations of a Complete and Backtrack-Free Interactive Configurator

Sathiamoorthy Subbarayan[1], Rune M. Jensen[1], Tarik Hadzic[1], Henrik R. Andersen[1], Henrik Hulgaard[2], and Jesper Møller[2]

[1] Department of Innovation, IT University of Copenhagen,
Rued Langgaards Vej 7, DK-2300 Copenhagen S, Denmark
{sathi,rmj,tarik,hra}@itu.dk
[2] Configit Software A/S,
Vermundsgade 38 B, DK-2100 Copenhagen Ø, Denmark
{jm,henrik}@configit-software.com

Abstract. A product configurator should be complete and backtrack free in the sense that the user can choose freely between any valid configuration and will be prevented from making choices that no valid configuration satisfies. In this paper, we experimentally evaluate a symbolic and search-based implementation of an interactive product configuration algorithm with these properties. Our results show that the symbolic approach often has several orders of magnitude faster response time than the search-based approach due to the precompilation of a symbolic representation of the solution space. Moreover, the difference between the average and worst response time for the symbolic approach is typically within a factor of two, whereas it for the search-based approach may be more than two orders of magnitude.

1 Introduction

Product configuration involves two major tasks: to define a modeling language that makes it easy to specify and maintain a formal product model and to develop a decision support tool that efficiently guides users to desirable product configurations [1]. In this paper we focus on the latter task and investigate a *complete* and *backtrack-free* approach to Interactive Product Configuration (IPC). The input to our IPC algorithm is a product model that consists of a set of product variables with finite domains and a set of product rules. A valid product configuration is an assignment to each variable that satisfies the product rules. The IPC algorithm initially has an empty set of user selected assignments. In each iteration of the algorithm, invalid values in the domain of the variables are pruned away such that each value in the domain of a variable is part of at least one valid configuration satisfying the existing set of assignments. The user then selects his preferred value for any free variable. The algorithm terminates when each variable in the product model is assigned a value. The IPC algorithm is complete in the sense that the user can choose freely between any valid configuration. It is also backtrack-free since the user is prevented from choosing a variable assignment for which no valid configuration exists. Hence the algorithm never forces the user to go back to a previous configuration state to explore alternative choices.

In this paper we experimentally evaluate a symbolic and search-based implementation of the IPC algorithm. The symbolic implementation is using a two-phase approach [2]. In the first phase, the product rules are compiled into a reduced ordered Binary Decision Diagram (BDD) [3] representing the set of valid configurations (the *solution space*). In the second phase, a fast specialized BDD operation is used to prune the variable domains. 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. The search-based implementation prunes variable domains by iterating over each possible assignment and searching for a valid configuration satisfying it. This approach can be improved by memorizing all assignments for which no solution exist across iterations and, within each iteration, adding all assignments in a found solution (not just the one being verified).

The symbolic approach has been implemented using *Configit Developer 3.2* [4] that employs a BDD-derived symbolic representation called Virtual Tables (VTs). The search-based approach has been implemented using *ILOG Solver 5.3* [5]. We have developed a publicly available benchmark suite called CLib [6] for the experiments that consists of 14 configuration problems. Our results show that the symbolic approach often has several orders of magnitude faster response time than the search based approach. In addition, the difference between average and worst response time is often within a factor of two for the symbolic approach, but some times more than two orders of magnitude for the search-based approach. Moreover, our results indicate that the configuration space of industrial configuration problems often have small symbolic representations. This result is somewhat surprising since BDDs blow up for many combinatorial problems investigated in AI (e.g., the n-queens problem and other permutation problems). It may be the frequent hierarchical structure of real-world configuration problems that make them particularly well-suited for BDDs.

The remainder of the paper is organized as follows. In Sect. 2, we define product configuration and describe the IPC algorithm. The symbolic and search-based implementation of the IPC algorithm are described in Sect. 3 and Sect. 4, respectively. We focus on describing the symbolic implementation since we assume that the search-based implementation is straight forward for most readers. Section 5 presents experimental work. Finally, we describe related work in Sect. 6 and draw conclusions in Sect. 7.

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 formulas defining which combinations are valid assignments. Each variable represents a product attribute. The variable domain refers to the options available for its attribute and formulas 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 $x_1, x_2, \cdots, x_n$, D is the Cartesian product of their finite domains $D_1 \times D_2 \times \cdots \times D_n$, and $F = \{f_1, f_2, \cdots, f_m\}$ is a set of propositional formulas over atomic propositions $x_i = v$, where $v \in D_i$, specifying conditions that the variable assignments must satisfy.*

Each formula f_i is a propositional expression inductively defined by

$$\phi \equiv x_i = v \mid \phi \wedge \psi \mid \phi \vee \psi \mid \neg\phi, \tag{1}$$

where $v \in D_i$. We use the abbreviation $\phi \Rightarrow \psi \equiv \neg\phi \vee \psi$ 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 the Boolean satisfiability problem can be reduced to it in polynomial time.

Example 1. 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 $f_1 = (x_3 = MIB) \Rightarrow (x_1 = black)$ and $f_2 = (x_3 = STW) \Rightarrow (x_2 \neq small)$. 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 Fig. 1. $\Diamond$

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

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

By interactive product configuration we refer 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 possible solutions to configurations consistent with this new condition. The user keeps selecting variable values until only one configuration is left. The IPC algorithm in Fig. 2 illustrates this interactive process. In line 1, the configurator takes a given configuration problem C and compiles

IPC(C)
1 $\quad R \leftarrow$ COMPILE(C)
2 $\quad$ **while** $|R| > 1$
3 $\quad\quad$ **do choose** $(x_i = v) \in$ VALID-ASSIGNMENTS(R)
4 $\quad\quad\quad R \leftarrow R|_{x_i = v}$

Fig. 2. The IPC Algorithm

it into an internal representation R. The procedure VALID-ASSIGNMENTS(R) in line 3 extracts the set of valid assignments (choices) from the internal representation. In line 4, the internal representation is restricted to configurations satisfying the new condition. 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. The symbolic and search-based implementation of the IPC algorithm differ in their internal representation and implementation of the valid assignment and restrict operations.

Example 2. For the T-shirt problem, the assignment $x_2 = \textit{small}$ will, by the second rule, imply $x_3 \neq \textit{STW}$ and since there is only one possibility left for variable x_3, it follows that $x_3 = \textit{MIB}$. The first rule then implies $x_1 = \textit{black}$. Unexpectedly, we have completely specified a T-shirt by just one assignment. ◊

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 a 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 Symbolic Implementation of the IPC Algorithm

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 and guarantees a response time that is polynomially bounded with respect to its size. The symbolic approach is offline to compile the configuration problem to a representation of the solution space 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, however, in general avoid exponentially large representations.

3.1 Symbolic Solution Space Representation

A configuration problem 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 binary as a vector of Boolean values $\boldsymbol{v} = (v_{l_i-1}, \cdots, v_1, v_0) \in \mathbb{B}^{l_i}$. Analogously, every variable x_i can be encoded by a vector of Boolean variables $\boldsymbol{b} = (b_{l_i-1}, \cdots, b_1, b_0)$. Now, the formula $x_i = v$ can be represented as a Boolean function given by the expression $b_{l_i-1} = v_{l_i-1} \wedge \cdots \wedge b_1 = v_1 \wedge b_0 = v_0$ (written $\boldsymbol{b} = \boldsymbol{v}$). For the T-shirt example we have, $D_2 = \{small, medium, large\}$ 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 and large as 10.

The translation to a Boolean domain is not surjective. There may exist assignments to the Boolean variables yielding invalid values. For example, the combination 11 does not encode a valid value in D_2. Therefore we introduce a *domain constraint* that forbids these unwanted combinations $F_D = \bigwedge_{i=1}^{n} (\bigvee_{v \in D_i} x_i = v)$. Furthermore, we define a translation function τ that maps a propositional expression ϕ to the Boolean function it represents

$$\tau(\phi) : \prod_{i=1}^{n} \mathbb{B}^{l_i} \to \mathbb{B}. \tag{2}$$

The translation is defined inductively as follows

$$\begin{aligned} \tau(x_i = v) &\equiv (\boldsymbol{b}_i = \boldsymbol{v}) && (3) \\ \tau(\phi \wedge \psi) &\equiv \tau(\phi) \wedge \tau(\psi) && (4) \\ \tau(\phi \vee \psi) &\equiv \tau(\phi) \vee \tau(\psi) && (5) \\ \tau(\neg \phi) &\equiv \neg \tau(\phi). && (6) \end{aligned}$$

Finally we can express a Boolean function representation $S'(C)$ of the solution space $S(C)$.

$$S'(C) \equiv \tau(F_D) \wedge \bigwedge_{j=1}^{m} \tau(f_j). \tag{7}$$

The resulting symbolic implementation of the IPC algorithm is shown in Fig. 3. In this implementation, the internal representation is a Boolean encoding *Sol* of the solution space. In order to restrict the solution space in line 4, it is conjoined with the Boolean encoding of the assignment chosen by the user.

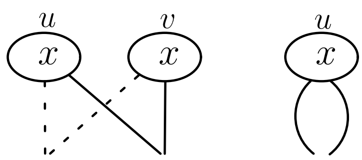
```
Symbolic-IPC(C)
    Sol ← S'(C)
    while |Sol| > 1
        do choose (x_i = v) ∈ Valid-Assignments(Sol)
            Sol ← Sol ∧ τ(x_i = v)
```

Fig. 3. Symbolic implementation of the IPC algorithm

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 is reduced such that no pair of distinct nodes u and v are associated with the same variable and low and high successors (Fig. 4a), and no variable node u has identical low and high successors (Fig. 4b). Due to these reductions,



(a) (b)

Fig. 4. (a) nodes associated to the same variable with equal low and high successors will be converted to a single node. (b) nodes causing redundant tests on a variable are eliminated. High and low edges are drawn with solid and dashed lines, respectively

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 [3]. 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 [3], 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 [3] and the books [7, 8].

3.3 BDD-Based Implementation of the IPC Algorithm

In the offline phase of BDD-based interactive configuration, we compile a BDD $\tilde{S}(C)$ of the Boolean function $S'(C)$ of the solution space. The variable ordering of Boolean variables 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

$$\tilde{\tau}(x_i = v) \equiv \text{ BDD of } \tau(x_i = v) \tag{8}$$
$$\tilde{\tau}(\phi \wedge \psi) \equiv \mathrm{Op}_{\wedge}(\tilde{\tau}(\phi), \tilde{\tau}(\psi)) \tag{9}$$
$$\tilde{\tau}(\phi \vee \psi) \equiv \mathrm{Op}_{\vee}(\tilde{\tau}(\phi), \tilde{\tau}(\psi)) \tag{10}$$
$$\tilde{\tau}(\neg\phi) \equiv \mathrm{Op}_{\neg}(\tilde{\tau}(\phi)). \tag{11}$$

In the base case (8), $\tilde{\tau}(x_i = v)$ denotes a BDD of the Boolean function $\tau(x_i = v)$ as defined in Sec. 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

$$\tilde{S}(C) \equiv \mathrm{Op}_{\wedge}(\tilde{\tau}(F_D), \tilde{\tau}(f_1), \cdots, \tilde{\tau}(f_m)). \tag{12}$$

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

Example 3. The BDD representing the solution space of the T-shirt example introduced in Sect. 2 is shown in Fig. 5. In the T-shirt example there are three variables: x_1, x_2 and x_3, whose domain sizes are four, three and two, respectively. As explained in Sect. 3.1, each variable is represented by a vector of Boolean variables. In the figure the Boolean vector for the variable x_i with domain D_i is $(x_i^{l_i-1}, \cdots, x_i^1, x_i^0)$, where $l_i = \lceil \lg |D_i| \rceil$. For example, in the figure, variable x_2 which corresponds to the size of the T-shirt is represented by the Boolean vector (x_2^1, x_2^0). In the BDD any path from the root node to the terminal node 1, corresponds to one or more valid configurations. For example, the path from the root node to the terminal node 1, with all the variables taking low values represents the valid configuration (*black*, *small*, *MIB*). Another path with x_1^1, x_1^0, and x_2^1 taking low values, and x_2^0 taking high value represents two valid configurations: (*black*, *medium*, *MIB*) and (*black*, *medium*, *STW*), namely. In this path the variable x_3^0 is a don't care variable and hence can take both low and high value, which leads to two valid configurations. Any path from the root node to the terminal node 0 corresponds to invalid configurations. $\Diamond$

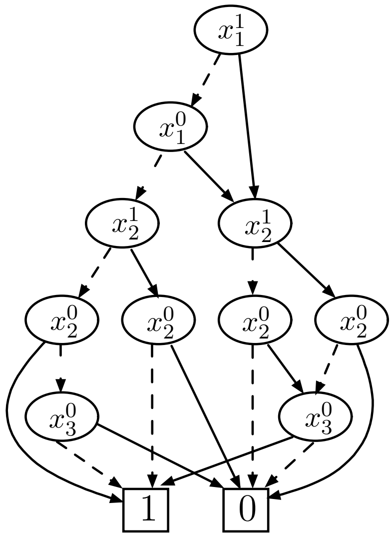


Fig. 5. BDD of the solution space of the T-shirt example. Variable x_i^j denotes bit b_j of the Boolean encoding of product variable x_i.

For a BDD version of the Symbolic-IPC algorithm, each Boolean operation is translated to its corresponding BDD operation. The response time 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-Assignments in Line 3. This procedure can be realized by a specialized BDD operation with worst-case complexity

$$O(\sum_{i=1}^{n} |V_i||D_i|), \tag{13}$$

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 Search-based Implementation of the IPC Algorithm

Search-based approaches like SAT techniques or CSP techniques can also be used to implement the IPC algorithm. In such cases the first step of the algorithm is to compile the configuration problem into an internal representation used by the applied search technique. The internal representation implicitly represents the set of valid configurations. The iterative step of the IPC algorithm is repeated until all the variables in the product model are assigned a value. For search-based techniques, the procedure for calculating VALID-ASSIGNMENTS is theoretically intractable. Every time an assignment is made, propagation is applied to prune the invalid values in the domain of all unselected variables. This can be done by checking whether each value in the domain of all the variables has at least one valid configuration satisfying the existing set of variable assignments. In the worst case, the search-based configurator has to find $1 - m + \sum_{i=1}^{m} d_i$ solutions for each iteration, where m is the number of unassigned variables during the iteration, and d_i is the domain size of the unassigned variable i during the iteration. Improved average complexity of VALID-ASSIGNMENTS can be obtained by memorizing invalid variable values. Since the solution space keeps decreasing, an invalid value will remain invalid in later iterations. In addition, within each iteration, whenever the search-based IPC finds a solution, it can mark all the assignments in the solution as valid. This can remove redundant search for solutions.

5 Experimental Evaluation

We use *Configit Developer 3.2* [4] for the symbolic implementation of the IPC algorithm. Configit Developer has a compiler which first checks for the semantic correctness of the product model. If the semantics is valid, it creates a Virtual Table (VT) that symbolically represents all valid solutions of the product model. For the search-based implementation of the IPC algorithm we use *ILOG Solver 5.3* [5]. ILOG Solver is a commercial constraint programming based C++ library for solving constraint satisfaction/optimization problems. In a comparative study, it has been shown to outperform a range of other CSP solvers [9]. The source code of our benchmarks for ILOG Solver has been constructed to the best of our knowledge. But we are not experts on the ILOG Solver technology and there may exist more efficient implementations. ILOG Configurator [5] is another product from the same company which uses ILOG Solver for configuration problems. ILOG Configurator, however, only does set bound propagation. This means that it may be possible for a user to choose assignments for which no valid configuration exists if the ILOG Configurator is used for interactive configuration.

For each configuration benchmark used in the experiments, 1000 random requests are generated using Configit Developer. There are several cycles of requests. Each cycle contains a sequence of requests in increasing order. Each request consists of a set of assignments to some of the variables in the product model. Each cycle of requests simulates an interactive configuration process of a product.

All the experiments were carried out on a Pentium-III 1 GHz machine with 2GB of memory, running Linux. The benchmarks used in the experiments are available in the CLib benchmark suite [6] in the Configit input language and the ILOG Solver source language.

The results are shown in Table 1. The first column in the table lists the name of the benchmark. The second column lists the amount of CPU time in seconds used by Configit Developer for generating the corresponding VT file. The third column lists the size of the generated VT file in Kilo Bytes (KB). As for BDDs (see Sect. 3), the variable ordering plays a crucial role in the size of the VT files generated. Unless specified, the default variable order was used in the experiments. The fourth column lists the number of solutions (#Sol) in the generated VT file. After generating a VT, the number of solutions represented by it can easily be counted. This is equivalent to the total number of valid configurations for the instance. In the subsequent four columns, the CPU time for Average response (Avg. RT) and Worst response (Wst. RT) are listed in seconds for both Configit Developer and ILOG Solver. The response times listed for Configit Developer include the time taken for the requests generation and writing the requests into a file. The corresponding times listed for ILOG Solver only include the time taken for reading the requests information from the file. Requests are generated as specified above.

Table 1. Experimental Results

Benchmark	**Virtual Table**			**Avg. RT (sec)**		**Wst. RT (sec)**	
	Time(sec)	**Size(KB)**	**#Sol**	**Configit**	**ILOG**	**Configit**	**ILOG**
Renault	460.00	1292	$2.8\text{x}10^{12}$	**0.1273**	489.29*	**0.240**	489.29*
Bike	0.45	22	$1.3\text{x}10^{8}$	**0.0005**	1.855	**0.010**	882.68
PC	0.89	24	$1.1\text{x}10^{6}$	**0.0007**	1.302	**0.010**	2.12
PSR	0.38	37	$7.7\text{x}10^{9}$	**0.0014**	2.398	**0.010**	486.12
Parity32_13	30.00	1219	$2.0\text{x}10^{8}$	0.0960	**0.061**	0.416	**0.24**
Big-PC	14.82	76	$6.2\text{x}10^{19}$	0.0012		0.010	
v1	5.67†	253	$8.2\text{x}10^{123}$	0.1620		0.320	
w1	56.52†	1347	$1.0\text{x}10^{89}$	0.0680		0.160	
ESVS	0.25	6.7	$3.5\text{x}10^{9}$	**0.0004**	0.059	**0.010**	0.14
FS	0.25	5.8	$2.4\text{x}10^{7}$	**0.0003**	0.036	**0.010**	0.21
FX	0.22	5.3	$1.2\text{x}10^{6}$	**0.0003**	0.029	**0.010**	0.10
Machine	0.14	6.7	$4.7\text{x}10^{8}$	**0.0004**	0.009	**0.010**	0.03
C169_FV	2.30 (144)	287	$3.2\text{x}10^{15}$	**0.0134**	0.195	**0.010**	28.77
C211_FS	6.93 (957)	370	$1.4\text{x}10^{67}$	**0.0219**	0.314	**0.020**	67.09
C250_FW	3.22 (111)	308	$1.2\text{x}10^{37}$	**0.0148**	0.203	**0.010**	38.98
C638_FVK	16.53 (1980)	534	$8.8\text{x}10^{123}$	**0.0385**	0.608	**0.050**	72.62

*For finding one solution only (i.e., not complete).

†The variable order file has been provided by Configit Software.

The Renault car configuration benchmark is described in [10]. Configit Developer takes 460 seconds to generate the VT file containing 2.8x10^{12} solutions for the Renault instance. But ILOG Solver takes 489 seconds just to solve the problem represented as a CSP instance. The average response time obtained for Renault by Configit Developer is 0.1273 second. Corresponding worst response time is 0.240 second.

The Bike and PC instances are configuration examples provided along with Configit Developer. They represent a bike and a personal computer configuration problem. The size of the PC and Bike instances in the Configit language is around 500 and 700 lines of code, respectively. In both cases, the average response time for Configit Developer is only a fraction of a millisecond. This is possible as the sizes of the corresponding VT files are very small. The corresponding worst response time for those two instances are 10 milliseconds only. But the average and worst response times for ILOG Solver are comparatively very high. The worst response time for ILOG Solver in case of Bike is 882.68 seconds, which is more than 400 times the corresponding average response time. In case of the PC example, there is not a large difference between the average and the worst response time of ILOG Solver. But still the values are high when compared to the corresponding values for Configit Developer. The PSR benchmark represents a power supply restoration problem modelled as a configuration instance. Further details about this problem is available in [11]. The performance of ILOG Solver and Configit Developer on the PSR instance is similar to those obtained for the Bike instance, with a large worst case response time by ILOG Solver. Parity32_13 represents a parity learning problem as a configuration instance. Information about this problem is available in [12]. In case of the Parity32_13 problem, ILOG Solver has better performance compared to Configit Developer. The difference is not large though. It is interesting that the VT files for Bike, PC, and PSR are compiled faster than the corresponding average response times given by ILOG Solver.

Big-PC represents the configuration problem of a personal computer with several additional features than those of the previous PC instance. The v1 and w1 instances represent real configuration problems of some of the customers of Configit Software. They are anonymized by renaming.[3] For these two instances, the corresponding variable order file was also provided by Configit Software. These instances have very large product models. For example, the Big-PC has 2500 lines of code in the Configit language. The w1 instance has more than 66,000 lines of code. It is very hard and error prone to convert them manually to ILOG Solver code. Hence for those instances Table 1 only has results obtained for Configit Developer. Even though these instances represent very large product models, Configit Developer has fast average and worst response times.

The ESVS, FS, FX, and Machine instances are from [13]. The first three instances represent screw compressor configuration problems. The last one represents parts of a real-world 4-wheeled vehicle configuration problem, anonymized by renaming. These four instances are easy compared to the previous 8 instances.

[3] Due to legal issues, v1 and w1 are not available in CLib.

This is reflected by their VT file sizes. The VT files for these instances are generated in a fraction of a second. In case of the FS instance, the VT file generation time is almost equal to the corresponding worst response time by ILOG Solver.

The last four instances are automotive product configuration problems from [14]. The original instances are in DIMACS CNF format. Good variable orders for these instances are obtained with the BuDDy BDD package [15] using the sift dynamic variable ordering heuristic. Configit Developer uses these variable orders for efficient VT file generation. The time taken by BuDDy to find a good variable order is listed in brackets in the second column of the table. For some large CNF files in [14], we did not do experiments as it took a lot of time to find good variable orders. For those instances, the average response time and the worst response time given by ILOG Solver are very large.

6 Related Work

Compilation techniques have also been studied for search techniques. In [16], 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. The MST is a polynomial-size structure. Operations on the MSTs, however, are of exponential time complexity. This may lead to long response times for an interactive configurator based on MSTs.

Acyclic constraint networks and the tree clustering algorithm [17, 18] represent a CSP solution space in a more compact way, organizing it as a tree of solved sub-problems. For extracting a solution, the generated structure offers polynomial time guarantees in the size of the 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 [19].

We are only aware of one other symbolic precompilation technique. In [10], 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. The BDD and automata approach to two-phase interactive configuration may perform equally well. A major advantage of using BDDs, however, is that this data structure has been studied intensely in formal verification for representing formal models of large systems [20, 21]. In particular, the variable ordering problem is well studied [7]. Furthermore a range of powerful software packages have been developed for manipulating BDDs [15, 22]. To our knowledge, automata compilation has not reached this level of maturity.

Previous work comparing symbolic and search-based approaches is very limited. Comparative studies of the SAT search engine zChaff [23] and BDD compilation show that neither approach has generally better performance than the other [24]. Similarly, a comparison of the Davis-Putnam procedure and BDDs shows complementary strengths of the two approaches rather than one dominating the other [25]. For interactive configuration, though, an important advantage of the BDD approach is that it is possible to compile the solution space prior to user interaction whereas search must be interleaved with user requests.

7 Conclusion

In this paper we have compared a symbolic and search-based implementation of a complete and backtrack-free interactive product configuration algorithm. Our experimental results show that the symbolic approach often has several orders of magnitude faster response time than the search based approach due to the precompilation of the solution space into a symbolic representation. In addition, the difference between average and worst response time is often much smaller for the symbolic approach than for the search-based approach. Our results indicate that BDD-derived representations often are small for real-world configuration instances. We believe that this may be due to the modular structure of configuration problems with a frequent hierarchical tree-like decomposition of dependencies that BDDs are particularly well-suited for.

We are currently working on an open source C++ Library for BDD-based interactive configuration for research and education purposes (CLab, [26]) and a comprehensive benchmark suite of industrial configuration problems (CLib, [6]). Future work includes developing specialized BDD operations to support both the compilation phase and interactive phase of product configuration.

Acknowledgments

We would like to thank ILOG for their elaborate answers to our user questions for ILOG Solver 5.3. We also thank Erik R. van der Meer for providing the T-shirt example.

References

1. Sabin, D., Weigel, R.: Product configuration frameworks - a survey. Intelligent Systems, IEEE **13** (1998) 42–49
2. Hadzic, T., Subbarayan, S., Jensen, R.M., Andersen, H.R., Møller, J., Hulgaard, H.: Fast backtrack-free product configuration using a precompiled solution space representation. In: Proceedings of the International Conference on Economic, Technical and Organizational aspects of Product Configuration Systems, DTU-tryk (2004) 131–138

3. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **8** (1986) 677–691
4. Configit Software A/S. http://www.configit-software.com (online)
5. ILOG. http://www.ilog.com (online)
6. CLib: Configuration benchmarks library. http://www.itu.dk/doi/VeCoS/clib/ (online)
7. Meinel, C., Theobald, T.: Algorithms and Data Structures in VLSI Design. Springer (1998)
8. Wegener, I.: Branching Programs and Binary Decision Diagrams. Society for Industrial and Applied Mathematics (SIAM) (2000)
9. Fernández, A.J., Hill, P.M.: A comparative study of eight constraint programming languages over the boolean and finite domains. Constraints **5** (2000) 275–301
10. Amilhastre, J., Fargier, H., Marquis, P.: Consistency restoration and explanations in dynamic CSPs-application to configuration. Artificial Intelligence **1-2** (2002) 199–234 ftp://fpt.irit.fr/pub/IRIT/RPDMP/Configuration/.
11. Thiébaux, S., Cordier, M.O.: Supply restoration in power distribution systems – a benchmark for planning under uncertainty. In: Pre-Proceedings of the 6th European Conference on Planning (ECP-01). (2001) 85–96
12. Parity-Function. ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/contributed/crawford/README (online)
13. Tiihonen, J., Soininen, T., Niemelä, I., Sulonen, R.: Empirical testing of a weight constraint rule based configurator. In: ECAI 2002 Configuration Workshop. (2002) 17–22 http://www.soberit.hut.fi/pdmg/Empirical/index.html.
14. Sinz, C., Kaiser, A., Kchlin, W.: Formal methods for the validation of automotive product configuration data. Artificial Intelligence for Engineering Design, Analysis and Manufacturing **17** (2003) 75–97 Special issue on configuration http://www-sr.informatik.uni-tuebingen.de/~sinz/DC/.
15. Lind-Nielsen, J.: BuDDy - A Binary Decision Diagram Package. http://sourceforge.net/projects/buddy (online)
16. Weigel, R., Faltings, B.: Compiling constraint satisfaction problems. Artificial Intelligence **115** (1999) 257–287
17. Dechter, R., Pearl, J.: Network-based heuristics for constraint-satisfaction problems. Artificial Intelligence **34** (1987) 1–38
18. Dechter, R., Pearl, J.: Tree-clustering schemes for constraint-processing. Artificial Intelligence **38** (1989) 353–366
19. Madsen, J.N.: Methods for interactive constraint satisfaction. Master's thesis, Department of Computer Science, University of Copenhagen (2003)
20. Burch, J.R., Clarke, E.M., McMillan, K.: Symbolic model checking: 10^{20} states and beyond. In: Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science. (1990) 428–439
21. Yang, B., Bryant, R.E., O'Hallaron, D.R., Biere, A., Coudert, O., Janssen, G., Ranjan, R.K., Somenzi, F.: A performance study of BDD-based model checking. In: Formal Methods in Computer-Aided Design FMCAD'98. (1998) 255–289
22. Somenzi, F.: CUDD: Colorado university decision diagram package. ftp://vlsi.colorado.edu/pub/ (1996)
23. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference (DAC'01). (2001)
24. Pan, G.and Vardi, M.Y.: Search vs. symbolic techniques in satisfiability solving. In: Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004). (2004)

25. Uribe, T.E., Stickel, M.E.: Ordered binary decision diagrams and the Davis-Putnam procedure. In Jouannaud, J.P., ed.: Proceedings of the 1st International Conference on Constraints in Computational Logics. Volume 845 of Lecture Notes in Computer Science., Springer (1994)
26. Jensen, R.M.: CLab: A C++ library for fast backtrack-free interactive product configuration. `http://www.itu.dk/people/rmj/clab/` (online)