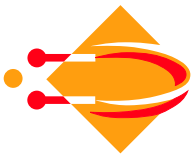

An introduction to solving interactive configuration problems

A Technical Report at the ITU

Tarik Hadzic
Henrik Reif Andersen

IT University Technical Report Series

TR-2004-49

ISSN 1600–6100 **August 2004**

ISSN 1600–6100

ISBN 87-7949-071-9

Copies may be obtained by contacting:

IT University of Copenhagen
Rued Langgaards Vej 7
DK-2300 Copenhagen S
Denmark

Telephone: +45 72 18 50 00
Telefax: +45 72 18 50 01
Web www.itu.dk

AN INTRODUCTION TO SOLVING INTERACTIVE CONFIGURATION PROBLEMS

TARIK HADZIC
HENRIK REIF ANDERSEN

Abstract. Configuration problems emerged as a research topic in the late 1980s as the result of manufacturing shift from mass-production to mass-customization. The essential part of a configuration problem is assembling the parts that satisfy given specifications. Several theoretical frameworks have attempted to formalize this core notion and each of them have facilitated many diverse solution techniques for handling different application areas.

In this paper we have concentrated on the increasingly important application area of *interactive configuration* which denotes a process of a user interactively specifying a product (a service). In particular, we explored the solution techniques using the frameworks of constraint satisfaction problems, binary decision diagrams and boolean satisfiability solving.

1. Introduction

What is configuration. Configuration problems have been recognized as topics of research since the 1980s. They have emerged from a change of industry orientation from mass-production toward mass-customization of products. In order to maintain the advantages of high volume production while meeting the demand for the increased needs for customization, configuration of products becomes an important issue.

There are many view points on configuration from the modelling of product knowledge to the development of efficient algorithms for ensuring intuitive and valid interactions. While some authors are developing adequate formalisms for describing knowledge-based configuration [25, 30, 4], others are exploring user-friendly requirements and underlying algorithms [22] or exploring the impact of configuration technology on existing business models.

Also, there are very different industrial applications of configuration in the companies' business processes. Not all configuration frameworks meet the demands equally well. Some of them are based on different theoretical concepts, not very fit for some functionalities, but still deliver satisfying functionality on other parameters.

One of the first attempts to define configuration in a domain independent way was by Mittal and Frayman in 1989 [21]. It describes configuration as a design activity of assembling an artifact that is made of a fixed set of well defined component types where components can interact only in predefined

Key words and phrases. configuration, constraint satisfaction, binary decision diagrams, satisfiability.

ways. Our formal definition captures this as a mathematical object with three elements: variables (often also called *parameters*), domains (or *values*) for the variables defining the combinatorial space of possible assignments, and formulas (or *rules*) defining which of the assignments are valid.

Definition 1. A *configuration problem* $\mathcal{C}$ is a triple $(\mathcal{X}, \mathcal{D}, \mathcal{F})$ where $\mathcal{X}$ is a set of variables $x_1, x_2, \ldots, x_n$, $\mathcal{D}$ a set of their finite domains $D_1, D_2, \ldots, D_n$ and $\mathcal{F} = \{f_1, f_2, \ldots, f_m\}$ 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.

Formulas $\mathcal{F}$ are given by the following syntax:

$$f ::= \quad x_i = v \mid f \wedge g \mid f \vee g \mid \neg f \mid f \Rightarrow g \mid f \Leftrightarrow g \quad (v \in D_i)$$

Each formula f_i induces a relation R_i over some subset of $\mathcal{X}$, containing exactly those tuples that satisfy f_i. The terms f_i and R_i will be used interchangeably. We will also use the term *valid configuration* or just *configuration* to describe an assignment (to all the variables) that satisfies all the formulas simultaneously.

Interactive configuration. We are going to focus on the problem of *interactive configuration* i.e. a process of a user interactively specifying a product (a car, PC) or a service (an insurance policy, airplane ticket) for his specific needs using the supporting software called a *configurator*.

The user makes a choice for some specific component of a product (a specific feature of a service), i.e. he assigns a value v from some domain D_i to a variable x_i. After that, the configurator makes calculations on what are valid choices for the other available (undecided) variables. The configurator deletes those values from the domains that will inevitably lead to violation of some of the formulas $\mathcal{F}$. If a domain for an undecided variable has only one value left, the configurator automatically assigns that value to the variable. The user keeps selecting variable values until he has completely specified the desired product (i.e. a valid assignment to all variables is reached).

Let us just note that configuration does not have to be interactive. One of the possible scenarios is that a user selects his preferences to a product, i.e., he assigns values to variables, and possibly gives priority to each choice. After that the configurator searches for a valid configuration based on user preferences, i.e. the product specification that is closest to the user preferences. This, so called *batch configuration* [26] has been implemented in a rule-based reasoning system described in [28].

User-friendly aspects of interactive configuration. Because of the interactive nature of the configuration process, the configurator's *response time* plays a crucial role in the user experience. If it is too slow, the system will be unpleasant to use. In fact, it is recommended that the response time should not be slower than 250 msec to ensure the interactive feeling of working in real-time (as advised by user interface designers, see e.g. [23]).

A user should not be forced to make choices in a predefined order. He should be able to make selections for any particular feature at any time. This allows him to concentrate first on those features that are most important to him.

While respecting these user-friendly requirements, the configurator has to deliver a number of functionalities. The most important one is calculating valid domains for undecided variables. We will name this as the *Calculate_Valid_Domains* (CVD) function. It should satisfy two properties:

Inference: Any value that is included in a calculated domain is extendible to a valid configuration. This implies that after selecting any value from a calculated domain, a user will be able to continue to a valid configuration.

Completeness: Any value that can be extended to a valid configuration should be included in a calculated domain. This implies that if there is a valid configuration, a user should be able to specify it.

As a consequence, calculated domains contain those and only those values that are part of a valid configuration. We will refer to these two properties as *completeness-of-inference* (CI).

Other important functionalities are *restoration* and *assisted conflict resolution*. Restoration refers to the functionality of a user undoing the choice for some already assigned variable with the configurator recalculating valid domains. Assisted conflict resolution allows a user to force an invalid choice for a variable. In response, he gets a minimal list of choices that need to be changed in order to restore consistency. Different authors have also identified a number of other functionalities [22] such as: automatic completion of a partial configuration or full information on the consequences of a choice (pricing, delivery). In this article we will describe and discuss implementations of some of the core functionalities, concentrating first of all on the CVD function.

The remainder of the paper is organized as follows. In Section 2, we explore concepts in the area of constraint satisfaction problems (CSPs), and use them to describe and implement CVD-function. In Section 3, we show how to encode a solution space symbolically as a binary decision diagram (BDD) and illustrate the underlying interactive configuration algorithm. In Section 4, we investigate the possible implementation of interactive configuration using Boolean satisfiability solving (SAT). Section 5 presents some of the other configuration related research. Finally in Section 6, we conclude and consider directions for future work.

2. Configuration as a Constraint Satisfaction Problem

Configuration is often viewed as a Constraint Satisfaction Problem (CSP). For more than a decade the CSP community has been developing formalisms to adequately describe essential features of configuration [10, 21, 25, 20]. It has also developed general algorithms that can be used to implement some configurator functionalities. However, most promising are CSP compilation techniques used to reduce complexity of interactive configuration.

2.1. **Classical CSP framework.** There are a lot of good surveys on classical CSP [16, 19, 24]. In this article we decided to base our CSP terminology on Edward Tsang's comprehensive overview "Foundations of constraint satisfaction" [29]. We will present some of the most important classical CSP concepts, that will later help us to describe more easily interactive configuration.

Definition 2. *Constraint satisfaction problem* is defined by a set of variables $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$, set of constraints $\mathcal{C} = \{C_1, C_2, \ldots, C_m\}$, and set of nonempty domains $\mathcal{D} = \{D_1, D_2, \ldots, D_n\}$ for each variable x_i. Each constraint C_i is defined over some subset $\{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\}$ of the variables $\mathcal{X}$ and specifies the allowed combinations of these variable values $C_i \subseteq D_{i_1} \times D_{i_2} \times \ldots \times D_{i_k}$. The goal is to find one or more assignments to the variables $\mathcal{X}$, satisfying all the constraints $\mathcal{C}$ simultaneously.

The (partial) assignment to a subset of variables $\{x_{a_1}, \ldots x_{a_r}\} \subseteq \mathcal{X}$, is denoted as ρ. The k-ary constraint C_i is called *relevant under* ρ if all it's variables are instantiated, i.e. if $\{x_{i_1}, \ldots x_{i_k}\} \subseteq \{x_{a_1}, \ldots x_{a_r}\}$. The assignments to all variables $\mathcal{X}$ that satisfy all constraints $\mathcal{C}$ simultaneously will be referred to as *solution tuples*. We will denote the set of all possible solution tuples $Sol \subseteq D_1 \times D_2 \times \ldots \times D_n$ as the *CSP solution space*. On the other hand, for the set of all possible assignments $S = D_1 \times D_2 \times \ldots \times D_n$ we will use the term *CSP search space*.

Most of CSP algorithms and theoretical concepts are designed only for CSP problems where the constraints are defined over maximum 2 variables. These CSP problems are called *binary CSPs*. If a CSP concept is also adequate for non-binary CSPs, then sometimes the prefix *"general"* is added to stress that fact. Therefore, both the terms *general CSP* and *CSP* are used to describe CSP problems that do allow non-binary constraints.

The CSP framework can uniformly express a wide range of problems from graph coloring and resource allocation to scheduling. This uniform view of various problems leads to development of general theoretical concepts and solving techniques. This help us to achieve deeper understanding of the problems and when combined with problem-specific features the general solving techniques can deliver satisfying solutions.

2.2. **Graph-related concepts.** A lot of research is directed on exploring graph-related concepts and their connections with CSP solving techniques. Namely, the complexity of solving a CSP problem is tightly connected with the topology of the related graphs. These concepts also present an important step to a more structured representation of CSP problems, facilitating more powerful solving techniques. Here we introduce some of the most relevant terminology.

Definition 3. The *constraint graph* of a binary CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ is an undirected graph where each node represents a variable, and each arc represents a constraint between variables represented by the end points.

Definition 4. The *constraint hypergraph* of a CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ is a hypergraph (V, E) in which each node represents a variable in $\mathcal{X}$ $(V = X)$, and each hyperedge represents a constraint in $\mathcal{C}$ $(E \subseteq Pow(V))$.

Definition 5. The *primal graph* of a CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ is an undirected graph in which each node represents a variable in $\mathcal{X}$, and for every pair of distinct nodes whose corresponding variables are involved in any k-constraint in $\mathcal{C}$ there is an edge between them.

Definition 6. The *dual graph* of a CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ is an undirected graph in which each constraint in $\mathcal{C}$ is represented by a node, and there is a labelled arc between any two nodes that share variables. The arcs are labelled by the shared variables.

If a constraint graph of a binary CSP has a structure of a *tree*, then a solution to the CSP problem can be found efficiently [8, 9]. This is a well explored concept, especially used for finding more than one solution, where more computational power is invested in transforming an original CSP to a tree-structured one.

2.3. **Problem Reduction.** Problem reduction is a CSP solving technique which transforms an original CSP to a problem that is easier to solve or is recognized as insolvable. This technique is especially useful in combination with CSP search algorithms.

Definition 7. A value from domain D_i is *redundant* if it is not the part of any solution tuple (satisfying assignment).

Definition 8. A k-tuple ($k \leq n$) over variables $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$ is *redundant* if it cannot be extended to any solution tuple.

The process of removing redundant values from variable domains and removing redundant tuples from constraints is called *problem reduction*.

Definition 9. A domain D_i is called *minimal* if it contains only those values that can be extended to a solution tuple with respect to the set of already made assignments ρ.

Most of the reduction algorithms usually operate on symbolic representation of constraints, and are therefore based on removing redundant values (Def. 7). Removing redundant tuples (Def. 8) requires explicit representation of constraint tuples, which can lead to a memory blow-up. Also note that the notion of minimal domain (Def. 9) is closely related to the inference-property of interactive configurator (page 3). We will now introduce the most important consistency concepts.

Definition 10. A CSP is *1-consistent* if every value v in every domain D_i satisfies all unary constraints on variable x_i. A CSP is *k-consistent* ($k \geq 2$) if for any assignment $\{x_{i_1} = v_1, x_{i_2} = v_2, \ldots, x_{i_{k-1}} = v_{k-1}\}$ satisfying all the relevant constraints (page 4), and for any additional variable x_{i_k} there exists a value $v_k \in D_{i_k}$ such that assignment $\{x_{i_1} = v_1, x_{i_2} = v_2, \ldots, x_{i_{k-1}} = v_{k-1}, x_{i_k} = v_k\}$ satisfies all the relevant constraints.

Because k-consistency does not imply $(k-1)$-consistency we introduce another term:

Definition 11. A CSP problem is *strongly k-consistent* if it is $1, 2, \ldots k-1, k$ consistent.

The most explored levels are 1-consistency and strong 2-consistency. They are equivalent to well studied *node consistency* and *arc consistency* for binary CSPs.

Definition 12. A binary CSP is *node-consistent* if and only if for all variables, all values in its domain satisfy the constraints on that variable.

Definition 13. An arc (x_i, x_j) in the constraint graph of a binary CSP is *arc-consistent* if and only if for every value v in the domain of x_i which satisfies the constraint on x_i, there exists a value in the domain of x_j which is compatible with $x_i = v$

A CSP is *arc-consistent* if and only if every arc in its constraint graph is arc-consistent. Some concepts for binary CSPs can be extended to general case if we introduce the two (equivalent) terms: *general arc consistency* and *hyper arc consistency*.

Definition 14. A k-ary constraint C is *generalized arc consistent* if the domains of all the variables involved in C are minimal with respect to C (Def. 9)

Definition 15. Constraint $C_i \subseteq D_{i_1} \times D_{i_2} \times \ldots \times D_{i_k}$ is *hyper-arc consistent* if for all $j = 1, \ldots, k$ and all $v \in D_{i_j}$ there exist a k-tuple $(v_1, \ldots, v_k) \in C_i$ such that $v_j = v$

2.4. **Search algorithms and other solving techniques.** Finding a satisfying assignment to all CSP variables is usually based on search algorithms. The basic search algorithm is called *chronological backtracking*. It performs depth-first search in CSP search space: Variables are instantiated sequentially. After each instantiation, all the relevant constraints are checked, and if there is any violation, backtracking is performed to the last instantiated variable that still has alternative values available.

This algorithm performs *thrashing*, ie. failing for the same reasons over and over again (in the same search spaces). Therefore, improvements to the algorithms were made so it could *learn from conflicts* and perform *intelligent backtracking*. These are well studied CSP concepts and are explained in most of the existing literature.

Another way to improve our backtracking search is to combine it with consistency algorithms to prune future search spaces. If after each assignment we remove those values from the remaining domains that contradict the assignment, we have a *forward checking algorithm*. If instead of just removing those values, we impose (general, hyper) arc-consistency on remaining domains we have a (general) *maintaining-arc-consistency* (MAC) algorithm. Actually, imposing completeness-of-inference in CVD-function is nothing more than imposing specific type of consistency which we will refer to as *completeness-of-inference (CI) consistency*. In that sense, our interactive configuration resembles running a sort of (general) MAC algorithm.

Unfortunately, all of these general CSP solving techniques have exponential worst-case complexity. In practice they might perform fast enough when we

are required to automatically find just one solution. However, when reasoning about all possible solutions, especially in interactive setting, this is too slow.

Beside reduction and search techniques, there is another approach that is especially well suited for finding all satisfying solutions. We call it the *solution synthesis* approach. It consists of gradually extending all non-redundant tuples (Def. 8) with new variable assignments that are not violating any constraint. At the end, we have a list of all complete assignments. However, this solving technique in addition to exponential time complexity has exponential space complexity as well.

2.5. **Implementing CVD-function.** Expressing the configuration problem $\mathcal{C}$ in CSP framework is a straight forward task. Variables and domains (in Def. 1 on page 2 and Def. 2 on page 4) mean exactly the same. Constraints C_i are identical to relations R_i induced by formulas f_i. The interactive-configuration process can be expressed as:

```
Naive Interactive-Configuration
  1:  completely_specified = FALSE
  2:  WHILE completely_specified = FALSE
  3:      DO CHOOSE x_i = v , (v ∈ D_i)
  4:      Calculate_Valid_Domains (x_i = v)
  5:      IF all variables x_i are assigned a value THEN
  6:      completely_specified = TRUE
```

In line 3, a user selects a value for one of the unassigned variables. He does so by choosing from currently valid domains. The computationally hard part is in line 4, where based on user selection ($x_i = v$) the configurator restricts domains for other unassigned variables, i.e. imposes completeness-of-inference (CI) consistency on remaining domains.

Implementation of CVD-function based on symbolic CSP-representation is rather naive and inefficient. After adding a new constraint $x_i = v$ we get a new CSP, denoted as $CSP_{x_i=v}$. The configurator now has to make each domain minimal (Def. 9) in $CSP_{x_i=v}$. This is an NP-hard task ([29]). In other words, every time a user selects a variable, we have to solve an NP-hard problem.

A way to overcome this NP-hardness is to transform the original CSP-problem to an equivalent one (having the same solution space) that enables *backtrack-free search*.

Definition 16. A search in CSP is *backtrack-free* under an ordering of variables if for every variable that is to be labelled, one can always find a value which is compatible with already labelled variables.

This means, that as long as we are assigning variable values that are satisfying all relevant constraints (involving only currently assigned variables), we are guaranteed that we will be able to continue to the complete valid assignment.

However, this is accomplished under fixed ordering of variables: $x_1 < x_2 < \ldots < x_n$, meaning that the user will not be able to choose the order in which he wants to make selections. This violates one of our user-friendly requirements described in chapter 1. Despite that, we will still explore how to utilize this

concept in possible implementations in order to illustrate how classical CSP concepts can be used for more efficient handling of interactive configuration.

A lot of research was carried on transforming CSPs to equivalent backtrack-free CSPs (mainly due to Freuder and Dechter during the 1980s). Transformations were accomplished either through imposing strong k-consistency where k is greater than the *width* of a primal graph, or by imposing *adaptive consistency* [29, 9]. This is a NP-hard problem, but it suffices to solve it only once. Subsequent CVD calls become polynomially fast.

Our interactive configuration reduces to:

```
Backtrack-free Interactive-Configuration
  1:  Transform CSP to a backtrack-free CSP.
  2:  k = 0
  3:  WHILE k < n
  4:        k = k + 1
  5:        DO CHOOSE x_k = v
  6:        Calculate_Valid_Domains (x_k = v)
```

When imposing CI-consistency (line 6), it is sufficient only to prune domain D_{k+1} by deleting only those values that are violating constraints involving $\{x_1, x_2, \ldots, x_k\}$. The task is now tractable and more adequate for interactive use. The NP-hard transformation in line 1 can be made *offline* leading to an idea of *compilation*.

2.6. **Compilation in CSP.** The *CSP compilation* refers to the computation of the "unchanging parts of constraint satisfaction problems into structures corresponding to a condensed representation of solution spaces" [30]. The choice of condensed representation (i.e. the data structure used for representing a solution space) varies depending on what kind of queries and transformations we want to perform on that data structure [4]. For example, the symbolic representation of constraints in a CSP problem is very condensed but does not support efficient queries and transformations (i.e. calculating valid domains). On the other hand, representation in form of the explicit enumeration of complete valid assignments allows efficient extraction of valid domains, but is not condensed and leads in all non-trivial instances to a memory blow-up.

Acyclic constraint networks and the *tree clustering algorithm* [9, 7] are used to represent CSP solution space in a more compact way, organizing it as a tree of solved subproblems. The generated structure offers polynomial time guarantees (in the size of the generated structure) for extracting valid domains. However, the size of the subproblems can not be controlled for all instances and would lead to exponential blow up. The complexity of the original problem is dominated by the complexities of subproblems - which are exponential in both space and time. Nevertheless, this is one of he first compilation approaches used to solve CSP problems. Additional compression of constraint networks by using cartesian product representation can improve the performance as indicated in [17].

Minimal synthesis trees [30] are also data structures used to compactly represent the set of all solutions in a CSP. It takes advantage of combining *consistency* techniques with *decomposition* techniques and *interchangeability* idea [30]. Unlike acyclic constraint networks that can be possibly exponentially large structures, while offering online polynomial time guarantees, the minimal synthesis trees are the polynomial-size structures that still require exponential time for finding the solution. The experimental evaluation indicates that this exponential worst-case rarely happens.

The compilation approach is not unique for the CSP community. The idea of representing the solution space in a more compact way in order to achieve better interaction algorithms can be treated as compilation from one language describing a problem to another language that supports some polynomial transformations and queries (like CVD function). This treatment within the *knowledge representation* community is reported in [4]. One of the described languages (data structures) that supports several desired poly-time queries and transformations is Binary Decision Diagrams.

3. Binary Decision Diagrams in Configuration

Binary Decision Diagrams (BDDs) is a data structure that has been widely used in the research community for symbolic computation, especially in the area of formal verification. BDDs came into focus of attention after Bryant's 1986 article [2]. They have proven to compactly represent large sets of satisfying assignments for many real-world problem instances in formal verification. In addition they allow certain poly-time queries and transformations [4] needed to efficiently implement CVD function. Therefore, they seem a reasonable choice for a data-structure representing the configuration solution space. Since there is not much work on using BDDs in interactive configuration, we will describe this approach in more details.

Definition 17 (Taken from [1])**.** A *Binary Decision Diagram (BDD)* is a rooted, directed, acyclic graph with

- one or two terminal nodes of out-degree zero labelled 1 and 0
- a set of variable nodes u of out-degree two, each labelled by a variable $var(u)$. The two outgoing edges are given by labelling children nodes with $low(u)$ and $high(u)$

A BDD is *Ordered* (OBDD) if on all paths through the graph the variables respect a given linear order $x_1 < x_2 < \ldots < x_n$. An OBDD is Reduced (ROBDD) if it satisfies the following properties:

Uniqueness: No two distinct nodes u and v have at any time the same variable name ($var(u)$) and low- and high-successor, ie. $var(u) = var(v) \wedge low(u) = low(v) \wedge high(u) = high(v) \Rightarrow u = v$

Non-redundant tests: No variable node u has an identical low- and high-successor, i.e. $low(u) \neq high(u)$

A Boolean function F defined over variables $x_1, x_2, \ldots, x_n$, can be encoded into a (RO)BDD structure. Each non-terminal node represents a variable,

and two outgoing edges represent a choice for the variable assignment (low(u): var(u) is assigned 0, high(u): var(u) is assigned 1). For example, the function $F(x_1, x_2) \equiv \neg(x_1 \Leftrightarrow x_2)$ is represented by the BDD in the Figure 1.

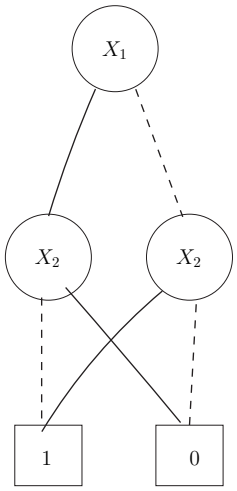

FIGURE 1. A decision tree for $\neg(x_1 \Leftrightarrow x_2)$. Dashed lines denote low-branches , solid lines high-branches.

ROBDD offers a *canonical representation* of any Boolean function with respect to a chosen variable ordering. This means that for any boolean function $f : B^n \to B$ ($B = \{0, 1\}$) there is exactly one ROBDD, with variable ordering $x_1 < x_2 < \ldots < x_n$, that is representing the function f.

Also, each non-terminal node has two outgoing arcs, and if we denote $|A|$ as the number of arcs in a ROBDD, and $|V|$ as a number of vertices, then a simple relation holds:

$$|A| = 2 \cdot (|V| - 2)$$

As a result, the problem of finding a satisfying assignment (complete configuration) has a $\Theta(|V|)$ complexity. The problem of checking if the formula is satisfiable can be reduced to checking that the graph is not just a single terminal node labelled 0 (constant time), and calculating a ROBDD that represents a Boolean function f induced by the assignment to a subset of its variables $f[x_1 = v_1, \ldots, x_m = v_m]$ ($v_i \in \{0, 1\}$) is also linear in the size of the original ROBDD for f. In addition, implementing general binary operator "*op*" on ROBDDs u_1 and u_2 representing functions f_1 and f_2 has the complexity $\Theta(|V_1| \cdot |V_2|)$, where $|V_1|, |V_2|$ are the numbers of vertices in u_1 and u_2 respectively. We will use f to denote both the Boolean function and the BDD representing that function [2], when there is no danger of creating confusion.

3.1. **Offline compilation.** As we mentioned, we will use ROBDDs to represent solution space of the original configuration problem. To do so, we first need to translate the finite-domain variables and formulas to Boolean-domain. Then, we encode the solution space of the equivalent Boolean-domain configuration problem to the ROBDD and during the interaction with a user, we translate

back the meanings of Boolean values to finite domain. We refer the reader to works of Hadzic et al. and Van der Meer [12, 18] for detailed description of this compilation approach. The translation to ROBDDs are based on Hu's work [14].

First, we need to encode finite domain variables $x_i \in X$ to Boolean variables. Each value $v \in D_i$ is mapped to a binary sequence $\overrightarrow{v} \in B^{N_i}$, where $N_i = \lceil \log |D_i| \rceil$. Then, for each variable x_i we introduce N_i Boolean variables $x_i^0, x_i^1 \ldots, x_i^{N_i-1}$, where variable x_i^k represents the k-th bit in the binary representation $\overrightarrow{v}$.

For example, for domain $D = \{0, 1, 2\}$ we have $N = \lceil \log 3 \rceil = 2$, so we could map $0 \in D$ to 00 ($x^0 = 0, x^1 = 0$), $1 \in D$ to 01 ($x^0 = 1, x^1 = 0$), and $3 \in D$ to 10 ($x^0 = 0, x^1 = 1$).

Now, we encode the propositional formulas $f_i \in \mathcal{F}$ over atomic propositions to *Boolean* functions $f_i^B \in \mathcal{F}^{\mathcal{B}}$. It is enough to replace each atomic proposition $x_i = v$ ($v \in D_i$) with $\overrightarrow{x_i} = \overrightarrow{v}$, ie. with $x_i^0 = v^0 \wedge x_i^1 = v^1 \wedge \ldots \wedge x_i^{N_i-1} = v^{N_i-1}$, $v^k \in \{0, 1\}$.

Finally, we have to impose additional restrictions on our Boolean variables since not every sequence of values v^k encodes a valid value $v \in D_i$. For example, the combination 11 does not encode a valid value in $D = \{0, 1, 2\}$. Therefore we explicitly forbid these combinations by adding *domain constraints*. A domain constraint in our example would be $f_D(x^0, x^1) \equiv \neg(x^0 = 1 \wedge x^1 = 1)$. Domain constraint for the entire configuration problem $\mathcal{C}(\mathcal{X}, \mathcal{D}, \mathcal{F})$ would be $f_{\mathcal{D}} = \bigwedge_{D \in \mathcal{D}} f_D$.

After translating to the Boolean domain, we fix a variable ordering and build a ROBDD representing the Boolean function $F \equiv \bigwedge_{f \in \mathcal{F}^{\mathcal{B}}} f \wedge f_{\mathcal{D}}$. The efficient way to build the ROBDD of a given Boolean function F (as presented in [14, 18]) is done by building small ROBDDs starting from terminal nodes, and incrementally building ROBDDs from existing ones by performing Boolean BDD operations with simultaneous enforcing of canonicity.

This building of ROBDD is the most sensitive part of the compilation process because the bad variable ordering can lead to a memory blow-up. But, since we are performing this step offline, we are usually able to fine tune the variable ordering, and overcome this NP-hard part of the problem.

3.2. **Online interaction.** After the generation of our ROBDD, we can start the interaction with the user.

```
BDD Interactive-Configuration
  1:  Generate ROBDD representing solution space.
  2:  completely_specified = FALSE
  3:  WHILE completely_specified = FALSE
  4:      DO CHOOSE x_i = v , (v ∈ D_i)
  5:      Calculate_Valid_Domains (x_i = v)
  6:      IF all variables x_i are assigned a value THEN
  7:      completely_specified = TRUE
```

Note that although the variable-order plays a significant role in generating the BDD, it does not enforce the order in which the user is making his choices.

The CVD-function in line 5 takes an assignment $x_i = v$, translates it into the appropriate conjunction of Boolean assignments $\overrightarrow{x_i} = \overrightarrow{v}$ and generates a new ROBDD $f' = f[\overrightarrow{x_i} = \overrightarrow{v}]$ (a linear-time operation).

Now the valid domain for an undecided variable x_k can be efficiently calculated by going through the list of all possible values $v \in D_k$ and checking whether the ROBDDs corresponding to the $x_k = v$ (i.e. the $f'[\overrightarrow{x_k} = \overrightarrow{v}]$) are reduced to the terminal node 0 (constant time). If the ROBDD is not just a terminal 0, the choice $x_k = v$ can be extended to a valid configuration and v is therefore added to the valid domain. Each assignment $f'[\overrightarrow{x_k} = \overrightarrow{v}]$ takes $O(|f'|)$ steps, and if we denote the k as the size of the largest domain in $\mathcal{D}$, n the number of variables in $\mathcal{X}$, then even the naive implementation takes $O(|f'| \cdot n \cdot k)$ steps which is linear in the size of the original ROBDD f'. Since translation between original (finite-domain) representation and Boolean representation is (time,space) efficient, the overall CVD-function takes polynomial time in the size of the ROBDD.

3.3. **Restoration and Assisted conflict resolution.** There are at least two ways to implement restoration. First, an easy trick that could also be implemented using other techniques is to memorize every intermediate ROBDD, and if the user decides to unassign the last decision, we simply switch to the previous ROBDD. The problem is that the user can unassign variables only in the predefined order (reverse to the assignment sequence).

Second, a more BDD-specific idea is to operate only with the original ROBDD f, and at each step to perform an entire set of assignments $f[x_1 = v_1, \ldots, x_m = v_m]$. This will not introduce significant rise in complexity since the assignment algorithm is efficient.

One simple way to implement assisted conflict resolution is after the user enforces conflict with invalid assignment $x_i = v_i$ in the sequence of assignments $x_1 = v_1, x_2 = v_2, \ldots, x_i = v_i$, to simply move the assignment $x_i = v_i$ to the beginning of the assignment queue ($x_i = v_i, x_1 = v_1, x_2 = v_2, \ldots, x_{i-1} = v_{i-1}$) and restart with the rest of the assignments until a conflict is reached ($x_i = v_i, x_1 = v_1, \ldots, x_k = v_k$). We remove the conflicting choice $x_k = v_k$ and continue until we reach another conflict or reach the last assignment $x_{i-1} = v_{i-1}$. At the end, we will have a list of choices that need to be changed in order to restore consistency. This implementation can be enhanced by applying a number of heuristics to the ordering of the sequence of remaining assignments (for example by giving preferences to which choice is more important to keep).

In this section, we have presented only the core user-friendly requirements, and we have described only the basic implementations. The BDD framework is robust enough to support a number of additional enhancements to these fundamentals. This is because the BDDs are investing more space in order to represent more information (the structure). This happens to be especially efficient when dealing with the structured instances (such as in a configuration domain). However, the robustness comes with the price of the large memory requirements, and possible compilation failures, when dealing with the unstructured problem instances. A possible solution to this problem might be to combine the BDD approach with other technologies that are able to handle large

numbers of variables without investing space in representing the structure. One of the most mature technologies with these properties is *boolean satisfiability solving* (SAT).

4. Using SAT in configuration

The satisfiability problem has both great theoretical and practical importance. It was the first problem shown to be NP-complete [3]. Today, significant performance improvements have been made to SAT-solving algorithms, and have found their way to applications in industry, from Automatic Test Pattern Generation to Theorem Proving and Verification. This performance efficiency of mature SAT techniques motivates us to explore its possible use in configuration. Since there is almost no work on this issue, and there are no experimental data to support any general conclusions, we will only describe the possible implementation and suggest some improvements to the basic model.

4.1. **Solving satisfiability problems.** Given a boolean formula F specified over variables $x_1, x_2, \ldots, x_n$, the *boolean satisfiability problem* (SAT) asks whether there exists an assignment to the variables such that the formula F evaluates to true.

The formula F is usually written in *conjunctive normal form* (CNF), which is a logical conjunction of *clauses* that are logical disjunctions of *literals*. A literal is a variable with or without negation. For example: $F = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$ represents a CNF form of the boolean formula F consisting of two clauses, over variables x_1, x_2, x_3.

The history of the SAT solving algorithms begins with a 1960 article from Davis and Putman [6], that suggested an algorithmic solution based on explicit resolution. In 1962, another article [5] from Davis, Logemann, Loveland proposed a solution based on backtracking-search, which is usually referred to as the DPLL algorithm, and represents the core of the state of the art SAT solvers today.

In the 1990s, there were some major advancements [11, 31, 27], mainly by introducing (conflict-driven) *learning* and *non-chronological backtracking*. Learning is a pruning technique that adds clauses to a clause database (and effectively forbids entering the same search space), while non-chronological backtracking means exactly the same as in a CSP community: backtrack is performed to a variable assignment that has caused the conflict.

We will now present a pseudo-code for a DPLL-based algorithm with learning, taken from Zhang et al. [31].

```
SAT solving algorithm
  1:  while(1) {
  2:  if (decide_next_branch()) {
  3:    while(deduce()==conflict) {
  4:       blevel = analyze_conflicts();
  5:       if (blevel < 0)
  6:       return UNSATISFIABLE;
  7:       else back_track(blevel);
  8:    }
```

```
9:  else //no branch means all variables got assigned.
10:  return SATISFIABLE;
11:  }
```

This listing compactly represents some key operations:

Branching: In the decide_next_branch() function, a new variable is chosen and a value (0 or 1) is assigned to it. If all the variables are already assigned a value, then the satisfying assignment has already been reached and the function returns *false*, effectively terminating the algorithm. If there are unassigned variables, the choice of a variable and the choice of a value is made based on a number of well explored variable-ordering and value-ordering heuristics. The function then returns *true* and proceeds to the deduction function.

Deducing: A *unit propagation* or *boolean constraint propagation* is performed, meaning that after assigning a value to the variable, some clauses might become one-literal clauses, and it is immediately deduced that this literal must evaluate to true which leads to the new assignment that in turn might generate a new one-literal clause etc... If the conflict is reached, the function proceeds to learning procedure in the analyze_conflicts() function. Actually, the DPLL based algorithm is performing a unit propagation most of the processing time, and it is essential to have efficient implementation of this function.

Learning: If the conflict is reached, a *conflict clause* is generated and added to a clause database. This is very useful for pruning the remaining search space of structured problems. In addition, the source of the conflict is found (in the form of the backtracking level) enabling the *intelligent backtracking*.

Backtracking: Backtracking means unassigning variables to the level of *conflicting assignment*. Different implementation techniques are proposed to accomplish restructuring of original clauses.

4.2. **Mapping configuration to SAT problem.** Translating a configuration problem to a SAT instance is quite similar to an already described approach for BDDs. The encoding to Boolean domain is similar. The only difference is that the function

$$F = \bigwedge_{f \in \mathcal{F}^{\mathcal{B}}} f \wedge f_{\mathcal{D}}$$

needs to be transformed to CNF instead to ROBDD.

An alternative approach would be to use multivalued propositional formulas $f \in \mathcal{F}$ from original configuration problem $\mathcal{C}$ instead of encoding to Boolean domains, creating

$$F = \bigwedge_{f \in \mathcal{F}} f$$

and transforming it to an instance of *multivalued SAT* (MV SAT). We would then be able to use the full power of well explored SAT techniques extended to work with finite domains. It would be reasonable to expect better performance compared to Boolean encoding, since both approaches take advantage of the

same SAT solving techniques, while the multivalued version does not involve adding domain constraints. However, this remains to be experimentally verified.

4.3. **Naive CNF-based implementation.** According to [4] the CNF langauge is too flat and does not support certain polytime queries and transformations. It is easy to see that CNF-based representation requires solving an NP-hard problem every time a user selects a variable (just as in the case of symbolic CSP representation).

Actually, without proper knowledge representation, the implementation of CVD-function reduces to solving a huge number of SAT instances, each without polynomial guarantees for response time. A naive implementation could look something like:

```
SAT based Interactive-Configuration
  1:  FOR all undecided variables x DO
  2:    FOR all values v ∈ D_x DO
  3:    IF F[x = v] is SATISFIABLE add v to D'_x
  4:    END FOR
  5:  END FOR
```

The $F[x = v]$ denotes a restriction of an original problem with an additional constraint $x = v$. D'_x denotes a valid domain for variable x.

The main strength of the SAT approach is the maturity of its technology that is performing extremely fast even when handling huge numbers of clauses and variables. It could be the case that although we do not have strong theoretical guarantees, the SAT solver is fast enough to provide a satisfying interactive performance. However, this performance should be tested on a number of instances before any conclusions can be drawn. Another advantage of the SAT approach is the extremely condensed representation of constraints. Significantly larger instances could be described using CNF-language instead of structured CSP or BDD representations.

There are several strategies we could use to improve this naive implementation. We could reduce complexity by avoiding redundant work via exploiting similarities among SAT instances using *incremental satisfiability* techniques [15, 13] or by storing complete (partial) solutions and reusing them for enhancing future searches. Still, there are no complexity guarantees.

We could also modify the user-friendly requirements by allowing the user to view valid domains one at a time. Unlike a backtrack-free search in a CSP community, were variables come in a predefined order, we offer the user to first choose the variable (component type), and then calculate the valid domain for that variable. The process of calculating the remaining valid domains can be continued in silent mode, while the user is exploring already available options. However, although significantly faster our function still has no response-time guarantees.

If the user has chosen variable x, and $|D_x| = k$, the calculation of the valid domain D'_x is reduced to solving k highly connected SAT instances. Exploiting incremental satisfiability could be very useful, especially if we take advantage

of the fact that only the value for x will be changed. We could encode it to variable ordering heuristics, so the last variable to be instantiated will be x.

4.4. **Compilation for SAT.** The CNF language obviously cannot give adequate theoretical performance guaranties. One of the reasons is, as we have already noticed, unability to represent and exploit structure of the original problem. For example, the efficiency of synthesis trees or BDDs is tightly related with this ability. We list a few ideas that could help in circumventing this theoretical inadequacy, trying to push more effort into the offline phase.

Divide to subproblems: Analogously to decomposition techniques for CSP (tree-clustering), we could exploit logical independencies among some variables. For example, we could identify subsets of highly related variables, transform constraints (clauses) to get independent subproblems, and improve efficiency by solving SAT subproblems.

Precompute variable and value heuristics: We could exploit topology of the problem by identifying those variables whose assignment could prune the most of the search space, and make these assignments earlier.

Transform SAT to more adequate structure: Certain classes of CNF formulas can be solved in polynomial time. This is the case if all the clauses have the length of 2 (literals) or if we are dealing with horn formulas.

5. Other configuration-related research

An important paradigm in configuration community is distinction between *configuration knowledge* and *configuration task*. The first term relates to capturing and representing information about components (variables), domains and rules. The latter term corresponds to manipulation of knowledge by executing user driven queries and transformations (which leads to specification of actual product). Our configuration knowledge was represented by synthesis trees, ROBDDs, and CNF formulas. Our configuration tasks were CVD-function, restoration and assisted conflict resolution.

Most of the research papers about configuration are dealing with configuration knowledge. They investigate representation models that will capture the essence of configuration problems. We have already mentioned some investigated properties: logical independence between certain variables (decomposition technique), interchangeability between variables (introducing meta variables). However, these are only consequences of the *hierarchical nature* of the configuration data. (we could look at synthesis trees as a way to capture some of that hierarchy). Researchers are extending existing frameworks and inventing new ones in order to naturally represent that hierarchy (good model supports faster processing algorithms).

The second aspect of configuration that we did not consider at all within our framework is called *dynamicity*. The *number of variables* that will be relevant to the final configuration *is not known in advance*. Different choices for the same component can require configuring different sets of other components (some motherboards in PC configuration support more functionalities than others).

There was a lot of research on extending CSP frameworks to capture dynamicity. This leads to introducing Dynamic CSPs [20] where *activity constraints* were introduced to reason about activity of the variables. Constraints will be checked only if all of the variables they are specifying are active. In [18] both of these aspects are investigated under the term of *modularity*.

There are also other configuration topics (dealing mostly with configuration knowledge). Some of the researchers are developing formalisms for describing knowledge bases. Others are exploring additional desirable properties that a knowledge representation should satisfy etc. It is a wide spectrum of topics and views on what configuration is and even a wider spectrum of implementations satisfying completely different user-friendly requirements and delivering different functionalities.

6. Conclusion

In this paper we have described interactive configuration problems and illustrated several approaches to solve them using the concepts and terminology from CSP, BDD and SAT community. In some cases we have moved from describing existing approaches to suggesting possible implementations. This was particulary the case for the SAT approach where applications to configuration were particulary rare.

Future work includes implementing interactive configuration using SAT-solvers. We hope to generate enough experimental data to evaluate the performance of SAT-based technology. Further, we plan to concentrate on exploring other data-structures and solving techniques in interactive configuration.

References

[1] Henrik Reif Andersen. An introduction to binary decision diagrams, 1999. Lecture notes for Efficient Algorithms and Programs, IT University of Copenhagen.

[2] Randy Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[3] Stephen A. Cook. The complexity of theorem proving procedures. *ACM Symposium on Theory of Computing*, 1971.

[4] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[5] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[6] Martin Davis and Hilary Putman. A computation procedure for quantification theory. *Journal of ACM*, 7:201–215, 1960.

[7] Rina Dechter. Constraint Networks. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, volume 1. Addison-Wesley Publishing Company, 1992. Second Edition.

[8] Rina Dechter and Judea Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.

[9] Rina Dechter and Judea Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366, 1989.

[10] Esther Gelle and Mihaela Sabin. Solving methods for conditional constraint satisfaction. In *Eighteenth International Joint Conference On Artificial Intelligence*, Workshop on Configuration, 2003.

[11] Evgueni Goldberg and Yakov Novikov. Berkmin: a fast and robust sat-solver. July 11 2002.

[12] Tarik Hadzic, Sathiamoorthy Subbarayan, Rune M. Jensen, Henrik R. Andersen, Jesper Moller, and Henrik Hulgaard. Fast backtrack-free product configuration using a precompiled solution space representation. In *Conference Proceedings of nternational Conference on Economic, technical and Organisational aspects of Product Configuration Systems*. Department of Manufacturing, Engineering and Management, Technical University of Denmark, June 2004.

[13] John N. Hooker. Solving the incremental satisfiability problem. *Journal of Logic Programming*, 15(1-2):177–18, January 1993.

[14] Alan John Hu. *Efficient Techniques for Formal Verification Using Binary Decision Diagrams*. PhD thesis, Stanford University, December 1995.

[15] Joonyoung Kim, Jesse Whittemore, and Karem Sakallah. On solving stack-based incremental satisfiability problems. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'00)*, pages 379–382, Washington - Brussels - Tokyo, September 2000. IEEE.

[16] Vipin Kumar. Algorithms for constraint-satisfaction problems: A survey. *The AI Magazine*, pages 32–44, 1992.

[17] Jeppe Madsen. Methods for interactive constraint satisfaction. Master's thesis, DIKU, February 2003.

[18] Erik Van Der Meer. *On Modular Configuration*. PhD thesis, IT University of Copenhagen, To appear.

[19] Ian Miguel and Qiang Shen. Solution techniques for constraint satisfaction problems: Foundations. *Artificial Intelligence Review*, 15(4):243–267, 2001.

[20] Sanjay Mittal and Brian Falkenhainer. Dynamic constraint satisfaction problems. In *Proceedings of the 8th National Conference on Artificial Intelligence*, pages 25–32, Hynes Convention Centre, July–August 1990. MIT Press.

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

[22] Bernard Pargamin. Vehicle sales configuration: the cluster tree approach. In *ECAI 2002 Configuration Workshop*, pages 35–40, 2002.

[23] Jef Raskin. *The Humane Interface*. Addison Wesley, 2000.

[24] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*, chapter 5. Constraint Satisfaction Problems, pages 137–160. Prentice Hall, second edition, 2003.

[25] Daniel Sabin and Eugene C. Freuder. Configuration as composite constraint satisfaction. In *Artificial Intelligence and Manufacturing Research Planning Workshop*, AAAI Technical Report, pages 28–36, 1996.

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

[27] Joao Marques Silva and Karem Sakallah. GRASP: A new search algorithm for satisfiability. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 220–227, Washington, November 10–14 1996. IEEE Computer Society Press.

[28] Carsten Sinz, Andreas Kaiser, and Wolfgang Kuchlin. Formal methods for the validation of automotive product configuration data. *Artificial Intelligence for Engineering Design*, 17:75–97, 2002.

[29] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London, 1993.

[30] Reiner Wiegel and Boi Faltings. Compiling constraint satisfaction problems. *Artificial Intelligence*, 115:257–287, 1999.

[31] Lintao Zhang, Sharad Malik, Matthew Moskewicz, and Conor Madigan. Efficient conflict driven learning in boolean satisfiability solver. In *Proceedings of the 2001 International Conference on Computer-Aided Design (ICCAD-01)*, pages 279–285, Los Alamitos, CA, November 4–8 2001. IEEE Computer Society.

E-mail address: `tarik@itu.dk, hra@itu.dk`