Calculating Valid Domains for BDD-Based Interactive Configuration

Tarik Hadzic, Rune Moller Jensen, Henrik Reif Andersen

Computational Logic and Algorithms Group,
IT University of Copenhagen, Denmark
tarik@itu.dk,rmj@itu.dk,hra@itu.dk

Abstract. In these notes we formally describe the functionality of Calculating Valid Domains from the BDD representing the solution space of valid configurations. The formalization is largely based on the Clab [1] configuration framework.

1 Introduction

Interactive configuration problems are special applications of Constraint Satisfaction Problems (CSP) where a user is assisted in interactively assigning values to variables by a software tool. This software, called a configurator, assists the user by calculating and displaying the available, valid choices for each unassigned variable in what are called *valid domains computations*. Application areas include customising physical products (such as PC's and cars) and services (such as airplane tickets and insurances).

Three important features are required of a tool that implements interactive configuration: it should be complete (all valid configurations should be reachable through user interaction), backtrack-free (a user is never forced to change an earlier choice due to incompleteness in the logical deductions), and it should provide real-time performance (feedback should be fast enough to allow real-time interactions). The requirement of obtaining backtrack-freeness while maintaining completeness makes the problem of calculating valid domains NP-hard. The real-time performance requirement enforces further that runtime calculations are bounded in polynomial time. According to user-interface design criteria, for a user to perceive interaction as being real-time, system response needs to be within about 250 milliseconds in practice [2]. Therefore, the current approaches that meet all three conditions use off-line precomputation to generate an efficient runtime data structure representing the solution space [3–6]. The challenge with this data structure is that the solution space is almost always exponentially large and it is NP-hard to find. Despite the bad worst-case bounds, it has nevertheless turned out in real industrial applications that the data structures can often be kept small [7, 5, 4].

2 Interactive Configuration

The input *model* to an interactive configuration problem is a special kind of Constraint Satisfaction Problem (CSP) [8, 9] where constraints are represented as propositional formulas:

Definition 1. *A* configuration model C *is a triple* (X, D, F) *where X is a set of variables* $\{x_0, \ldots, x_{n-1}\}$*,* $D = D_0 \times \ldots \times D_{n-1}$ *is the Cartesian product of their finite domains* $D_0, \ldots, D_{n-1}$ *and* $F = \{f_0, ..., f_{m-1}\}$ *is a set of propositional formulae over atomic propositions* $x_i = v$*, where* $v \in D_i$*, specifying conditions on the values of the variables.*

Concretely, every domain can be defined as $D_i = \{0, \ldots, |D_i| - 1\}$. An assignment of values $v_0, \ldots, v_{n-1}$ to variables $x_0, \ldots, x_{n-1}$ is denoted as an assignment $\rho = \{(x_0, v_0), \ldots, (x_{n-1}, v_{n-1})\}$. Domain of assignment $dom(\rho)$ is the set of variables which are assigned: $dom(\rho) = \{x_i \mid \exists v \in D_i.(x_i, v) \in \rho\}$ and if $dom(\rho) = X$ we refer to ρ as a *total assignment*. We say that a total assignment ρ is *valid*, if it satisfies all the rules which is denoted as $\rho \models F$.

A partial assignment $\rho', dom(\rho') \subseteq X$ is *valid* if there is at least one total assignment $\rho \supseteq \rho'$ that is valid $\rho \models F$, i.e. if there is at least one way to successfully finish the existing configuration process.

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

2.1 User Interaction

Configurator assists a user interactively to reach a valid product specification, i.e. to reach total valid assignment. The key operation in this interaction is that of computing, for each unassigned variable $x_i \in X \setminus dom(\rho)$, the *valid domain* $D_i^\rho \subseteq D_i$. The domain is *valid* if it contains those and only those values with which ρ can be extended to become a total valid assignment, i.e. $D_i^\rho = \{v \in D_i \mid \exists \rho' : \rho' \models F \wedge \rho \cup \{(x_i, v)\} \subseteq \rho'\}$.

The significance of this demand is that it guarantees the user backtrack-free assignment to variables as long as he selects values from valid domains. This reduces cognitive effort during the interaction and increases usability.

At each step of the interaction, the configurator reports the valid domains to the user, based on the current partial assignment ρ resulting from his earlier choices. The user then picks an unassigned variable $x_j \in X \setminus dom(\rho)$ and selects a value from the calculated valid domain $v_j \in D_j^\rho$. The partial assignment is then extended to $\rho \cup \{(x_j, v_j)\}$ and a new interaction step is initiated.

3 BDD Based Configuration

In [5, 10] the interactive configuration was delivered by dividing the computational effort into an *offline* and *online* phase. First, in the offline phase, the authors compiled a BDD representing the solution space of all valid configurations $Sol = \{\rho \mid \rho \models F\}$. Then, the functionality of *calculating valid domains* (CVD) was delivered online, by efficient algorithms executing during the interaction with a user. The benefit of this approach is that the BDD needs to be compiled only once, and can be reused for multiple user sessions. The user interaction process is illustrated in Fig. 2.

$InCo(Sol, \rho)$
1: `while` $|Sol^\rho| > 1$
2: `compute` $D^\rho = CVD(Sol, \rho)$
3: `report` D^ρ `to the user`
4: `the user chooses` (x_i, v) `for some` $x_i \notin \mathrm{dom}(\rho)$, $v \in D_i^\rho$
5: $\rho \leftarrow \rho \cup \{(x_i, v)\}$
6: `return` ρ

Fig. 2. Interactive configuration algorithm working on a BDD representation of the solutions *Sol* reaches a valid total configuration as an extension of the argument ρ.

Important requirement for online user-interaction is the guaranteed real-time experience of user-configurator interaction. Therefore, the algorithms that are executing in the online phase must be provably efficient in the size of the BDD representation. This is what we call the *real-time guarantee*. As the CVD functionality is NP-hard, and the online algorithms are polynomial in the size of generated BDD, there is no hope of providing polynomial size guarantees for the worst-case BDD representation. However, it suffices that the BDD size is small enough for all the configuration instances occurring in practice [10].

3.1 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. 3a), and no variable node u has identical low and high successors (Fig. 3b). Due to these reductions, the number of nodes

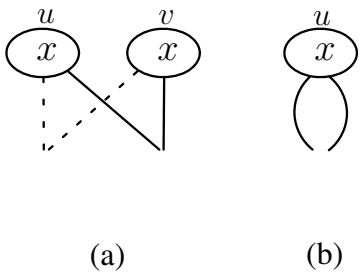

Fig. 3. (a) nodes associated to the same variable with equal low and high successors will be converted to a single node. (b) nodes causing redundant tests on a variable are eliminated. High and low edges are drawn with solid and dashed lines, respectively

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 [11]. 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 [11], 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 [11] and the books [12, 13].

3.2 Compiling the Configuration Model

Each of the finite domain variables x_i with domain $D_i = \{0, \ldots, |D_i| - 1\}$ is encoded by $k_i = \lceil log|D_i| \rceil$ Boolean variables $x_0^i, \ldots, x_{k_i-1}^i$. Each $j \in D_i$, corresponds to a

binary encoding $\overline{v_0 \dots v_{k_i-1}}$ denoted as $v_0 \dots v_{k_i-1} = enc(j)$. Also, every combination of Boolean values $v_0 \dots v_{k_i-1}$ represents some integer $j \leq 2^{k_i} - 1$, denoted as $j = dec(v_0 \dots v_{k_i-1})$. Hence, atomic proposition $x_i = v$ is encoded as a Boolean expression $x_0^i = v_0 \wedge \dots \wedge x_{k_i-1}^i = v_{k_i-1}$. In addition, *domain constraints* are added to forbid those assignments to $v_0 \dots v_{k_i-1}$ which do not translate to a value in D_i, i.e. where $dec(v_0 \dots v_{k_i-1}) \geq |D_i|$.

Let the solution space Sol over ordered set of variables $x_0 < \dots < x_{k-1}$ be represented by a Binary Decision Diagram $B(V, E, X_b, R, var)$, where V is the set of nodes u, E is the set of edges e and $X_b = \{0, 1, \dots, |X_b| - 1\}$ is an ordered set of variable indexes, labelling every non-terminal node u with $var(u) \leq |X_b| - 1$ and labelling the terminal nodes T_0, T_1 with index $|X_b|$. Set of variable indexes X_b is constructed by taking the union of Boolean encoding variables $\bigcup_{i=0}^{n-1} \{x_0^i, \dots, x_{k_i-1}^i\}$ and ordering them in a natural layered way, i.e. $x_{j_1}^{i_1} < x_{j_2}^{i_2}$ iff $i_1 < i_2$ or $i_1 = i_2$ and $j_1 < j_2$.

Every directed edge $e = (u_1, u_2)$ has a starting vertex $u_1 = \pi_1(e)$ and ending vertex $u_2 = \pi_2(e)$. R denotes the root node of the BDD.

Example 2. The BDD representing the solution space of the T-shirt example introduced in Sect. 2 is shown in Fig. 4. 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. 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 $(x_i^0, x_i^1, \cdots x_i^{l_i-1})$, 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_2^0, x_2^1). 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_1^0, x_1^1, and x_2^0 taking low values, and x_2^1 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. $\Diamond$

4 Calculating Valid Domains

Before showing the algorithms, let us first introduce the appropriate notation. If an index $k \in X_b$ corresponds to the $j+1$-st Boolean variable x_j^i encoding the finite domain variable x_i, we define $var_1(k) = i$ and $var_2(k) = j$ to be the appropriate mappings. Now, given the BDD $B(V, E, X_b, R, var)$, V_i denotes the set of all nodes $u \in V$ that are labelled with a BDD variable encoding the finite domain variable x_i, i.e. $V_i = \{u \in V \mid var_1(u) = i\}$. We think of V_i as defining a layer in the BDD. We define In_i to be the set of nodes $u \in V_i$ reachable by an edge originating from outside the V_i layer, i.e. $In_i = \{u \in V_i \mid \exists (u', u) \in E.\ var_1(u') < i\}$. For the root node R, labelled with $i_0 = var_1(R)$ we define $In_{i_0} = V_{i_0} = \{R\}$.

We assume that in the previous user assignment, a user fixed a value for a finite domain variable $x = v, x \in X$, extending the old partial assignment ρ_{old} to the current

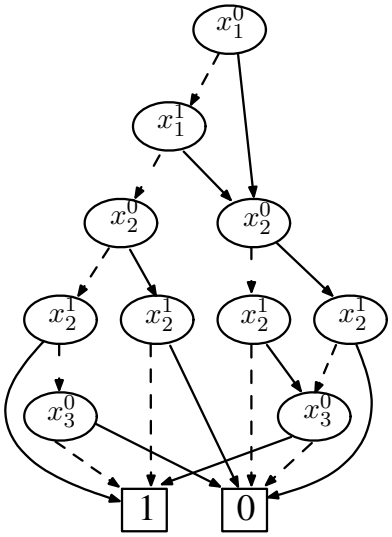

Fig. 4. BDD of the solution space of the T-shirt example. Variable x_i^j denotes bit v_j of the Boolean encoding of finite domain variable x_i.

assignment $\rho = \rho_{old} \cup \{(x, v)\}$. For every variable $x_i \in X$, old valid domains are denoted as $D_i^{\rho_{old}}, i = 0, \ldots, n-1$. and the old BDD $B^{\rho_{old}}$ is reduced to the restricted BDD, $B^{\rho}(V, E, X_b, var)$. The CVD functionality is to calculate valid domains D_i^{ρ} for remaining unassigned variables $x_i \notin dom(\rho)$ by extracting values from the newly restricted BDD $B^{\rho}(V, E, X_b, var)$.

To simplify the following discussion, we will analyze the isolated execution of the CVD algorithms over a given BDD $B(V, E, X_b, var)$. The task is to calculate valid domains VD_i from the starting domains D_i. The user-configurator interaction can be modelled as a sequence of these executions over restricted BDDs B^{ρ}, where the valid domains are D_i^{ρ} and the starting domains are $D_i^{\rho_{old}}$.

The CVD functionality is delivered by executing two algorithms presented in Fig. 5 and Fig. 6. The first algorithm is based on the key idea that if there is an edge $e = (u_1, u_2)$ crossing over V_j, i.e. $var_1(u_1) < j < var_1(u_2)$ then we can include all the values from D_j into a valid domain $VD_j \leftarrow D_j$.

We refer to e as a *long edge* of length $var_1(u_2) - var_1(u_1)$. Note that it skips $var(u_2) - var(u_1)$ Boolean variables, and therefore compactly represents the part of a solution space of size $2^{var(u_2)-var(u_1)}$.

For the remaining variables x_i, whose valid domain was not copied by $CVD - Skipped$, we execute $CVD(B, x_i)$ from Fig. 6. There, for each value j in a domain D_i' we check whether it can be part of the domain D_i. The key idea is that if $j \in D_i$ then there must be $u \in V_i$ such that traversing the BDD from u with binary encoding of j

```
CVD − Skipped(B)
1: for each i = 0 to n − 1
2:    L[i] ← i + 1
3: T ← TopologicalSort(B)
4: for each k = 0 to |T| − 1
5:    u1 ← T[k], i1 ← var1(u1)
6:    for each u2 ∈ Adjacent[u1]
7:       L[i1] ← max{L[i1], var1(u2)}
8: S ← {}, s ← 0
9: for i = 0 to n − 2
10:   if i + 1 < L[s]
11:      L[s] ← max{L[s], L[i + 1]}
12:   else
13:      if s + 1 < L[s]   S ← S ∪ {s}
14:      s ← i + 1
15: for each j ∈ S
16:   for i = j to L[j]
17:   VDi ← Di
```

Fig. 5. In lines 1-7 the $L[i]$ array is created to record longest edge $e = (u_1, u_2)$ originating from the V_i layer, i.e. $L[i] = max\{var_1(u') \mid \exists(u, u') \in E.var_1(u) = i\}$. The execution time is dominated by $TopologicalSort(B)$ which can be implemented as depth first search in $O(|E| + |V|) = O(|E|)$ time. In lines 8-14, the overlapping long segments have been merged in $O(n)$ steps. Finally, in lines 15-17 the valid domains have been copied in $O(n)$ steps. Hence, the total running time is $O(|E| + n)$.

```
CVD(B, xi)
1:  VDi ← {}
2:  for each j = 0 to |Di| − 1
3:     for each k = 0 to |Ini| − 1
4:     u ← Ini[k]
5:     u' ← Traverse(u, j)
6:     if u' ≠ T0
7:        VDi ← VDi ∪ {j}
8:        Return
```

Fig. 6. Classical CVD algorithm. $enc(j)$ denotes the binary encoding of number j to k_i values $v_0, \ldots, v_{k_i-1}$. If $Traverse(u, j)$ from Fig. 7 ends in a node different then T_0, then $j \in VD_i$.

will lead to a node other than T_0, because then there is at least one satisfying path to T_1 allowing $x_i = j$.

```
Traverse(u, j)
1:  i ← var_1(u)
2:  v_0, ..., v_{k_i - 1} ← enc(j)
3:  s ← var_2(u)
4:  if Marked[u] = j return T_0
5:  Marked[u] ← j
6:  while s ≤ k_i - 1
7:    if var_1(u) > i return u
8:    if v_s = 0 u ← low(u)
10:   else u ← high(u)
12:   if Marked[u] = j return T_0
13:   Marked[u] ← j
14:   s ← var_2(u)
```

Fig. 7. For fixed $u \in V, i = var_1(u)$, $Traverse(u, j)$ iterates through V_i and returns the node in which the traversal ends up.

When traversing with $Traverse(u, j)$ we mark the already traversed nodes u_t with j, $Marked[u_t] \leftarrow j$ and prevent processing them again in the future j-traversals $Traverse(u', j)$. Namely, if $Traverse(u, j)$ reached T_0 node through u_t, then any other traversal $Traverse(u', j)$ reaching u_t must as well end up in T_0. Therefore, for every value $j \in D_i$, every node $u \in V_i$ is traversed at most once, leading to worst case running time complexity of $O(|V_i| \cdot |D_i|)$. Hence, the total running time for all variables is $O(\sum_{i=0}^{n-1} |V_i| \cdot |D_i|)$.

The total worst-case running time for the two CVD algorithms is therefore $O(\sum_{i=0}^{n-1} |V_i| \cdot |D_i| + |E| + n) = O(\sum_{i=0}^{n-1} |V_i| \cdot |D_i| + n)$.

References

1. Jensen, R.M.: CLab: A C++ library for fast backtrack-free interactive product configuration. `http://www.itu.dk/people/rmj/clab/` (online)
2. Raskin, J.: The Humane Interface. Addison Wesley (2000)
3. Amilhastre, J., Fargier, H., Marquis, P.: Consistency restoration and explanations in dynamic CSPs-application to configuration. Artificial Intelligence **1-2** (2002)
4. Madsen, J.N.: Methods for interactive constraint satisfaction. Master's thesis, Department of Computer Science, University of Copenhagen (2003)
5. 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: PETO Conference, DTU-tryk (2004)
6. Møller, J., Andersen, H.R., Hulgaard, H.: Product configuration over the internet. In: Proceedings of the 6th INFORMS. (2004)
7. Configit Software A/S. `http://www.configit-software.com` (online)
8. Tsang, E.: Foundations of Constraint Satisfaction. Academic Press (1993)

9. Dechter, R.: Constraint Processing. Morgan Kaufmann (2003)
10. Subbarayan, S., Jensen, R.M., Hadzic, T., Andersen, H.R., Hulgaard, H., Møller, J.: Comparing two implementations of a complete and backtrack-free interactive configurator. In: CP'04 CSPIA Workshop. (2004) 97–111
11. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **8** (1986) 677–691
12. Meinel, C., Theobald, T.: Algorithms and Data Structures in VLSI Design. Springer (1998)
13. Wegener, I.: Branching Programs and Binary Decision Diagrams. Society for Industrial and Applied Mathematics (SIAM) (2000)