

# Integrating CSP Decomposition Techniques and BDDs for Compiling Configuration Problems

Sathiamoorthy Subbarayan

IT University of Copenhagen, Rued Langgaards Vej 7, DK-2300 Copenhagen S,  
Denmark  
sathi@itu.dk

**Abstract.** In this project we present, the tree-of-BDDs approach, a decomposition scheme for compiling configuration problems. We also present efficient techniques for generating minimum explanations in the BDD-based configuration schemes. Given a BDD representing the solutions of a CSP, an efficient technique for detecting full interchangeable values is given. Experimental results show that the techniques presented here drastically reduces the time and space requirements for interactive configurators.

## 1 Introduction

The complexity of made-to-order products keeps increasing. Examples of such products include personal computers, bikes, and power-backup systems. Such products will be represented in the form of a product model. A product model will list the number of parameters (variables) defining the product, their possible values and the rules by which those parameter values can be chosen. A product model implicitly represents all valid configurations of a product, and it can be viewed as a Constraint Satisfaction Problem (CSP), where the solutions to the CSP are equivalent to valid configurations of the corresponding product model. The increase in the complexity of made-to-order products rises the need for efficient decision support systems to configure a product based on the requirements posed by the customer. Such decision support systems are called configurators. Configurators read a product model which represents all valid configurations of the product and guides the user in choosing one among the valid configurations as close as possible to his requirements. An interactive configurator takes a product model as input and interactively helps the user to choose his preferred values for the parameters in the product model, one-by-one. The interactive configurator needs to be complete and backtrack-free. Complete means that all valid configurations need to be configurable using the configurator. Backtrack-Free means that the configurator should not allow the user to choose a value for a parameter in the product model which would eventually lead him to no valid configurations. To ensure backtrack-freeness, the interactive configurator needs to prune away values from the possible values of parameters in the product model as and when those values will not lead to any valid configuration. In addition,

the interactive configurator needs to give responses in a short period of time. Examples for commercial interactive configurators include Configit Developer[1] and Array Configurator[2].

The Binary Decision Diagram (BDD) [3] based symbolic CSP compilation technique [4, 5] can be used to compile all solutions of a configuration problem into a single (monolithic) BDD. Once a BDD is obtained, the functions required for interactive configuration can be efficiently implemented. The problem with such approaches is that they do not exploit the fact that configuration problems are specified in hierarchies. Due to this the BDD obtained after compilation could be unnecessarily large. Such hierarchies are closer to trees in shape. Hence, the tree decomposition techniques for CSPs could be used to enhance the compilation schemes for configuration problems. In this report, we present the tree-of-BDDs approach: a combination of the BDD-based compilation technique and a CSP decomposition technique for efficient compilation of all solutions. Precisely, the contributions of this project are

1. A compilation scheme for configuration problems, using the hinge CSP decomposition technique [6] and BDDs. In the experiments the scheme results in upto 96% reduction in space. Response time for interactions also decrease.
2. Efficient minimum explanation algorithms for two BDD-based compilation schemes.
3. Method to detect full interchangeable (FI) value sets in two BDD-based compilation schemes. The FI value sets can be used for problem reformulations.
4. Efficient implementation and experimental evaluation of the above techniques.

The drastic reduction in space requirement is of great importance in online configuration applications, where one needs to send the configuration details through the internet, and also in embedded configuration, where one needs to embed the configuration details on a product itself, so that it could be reconfigured as and when required. Reduction in response times enable the tree-of-BDDs scheme to scale high.

In Section 2, the basic definitions are given. In Section 3, the interactive configuration algorithm is listed. Representing solution space using BDDs is discussed in Section 4. The tree-of-BDDs compilation scheme, and the algorithms for explanations and detection of full interchangeable values are presented in the subsequent sections. Experimental results are presented in Section 8. Discussion on related work, followed by concluding remarks, finish this report.

## 2 Background

In this section we give the necessary background. Some of the definitions in this section are based on the definitions in [5, 7, 8].

**Definition 1.** *Let  $X$  be a set of variables  $\{x_1, x_2, \dots, x_n\}$  and  $D$  be the set  $\{D_1, D_2, \dots, D_n\}$ , where  $D_i$  is the domain of values for variable  $x_i$ .*

**Definition 2.** A relation  $R$  over the variables in  $M$ ,  $M \subseteq X$ , is a set of allowed combinations of values for the variables in  $M$ . Let  $M = \{x_{m1}, x_{m2}, \dots, x_{mk}\}$ , then  $R \subseteq (D_{m1} \times D_{m2} \times \dots \times D_{mk})$ .  $R$  restricts the ways in which the variables in  $M$  could be assigned values.

**Definition 3.** A constraint satisfaction problem instance *CSP* is a triplet  $(X, D, C)$ , where  $C = \{c_1, c_2, \dots, c_m\}$  is a set of constraints. Each constraint,  $c_i$ , is a pair  $(S_i, R_i)$ , where  $S_i \subseteq X$  is the scope of the constraint and  $R_i$  is a relation over the variables in  $S_i$ .

Without loss of generality, we assume that the variables whenever grouped in a set are ordered in a fixed sequence. The same ordering is assumed on any set of the values of variables and the pairs with variables in them.

**Definition 4.** An assignment is a pair  $(x_i, v)$ , where  $x_i \in X$  and  $v \in D_i$ . The assignment  $(x_i, v)$  bounds the value of  $x_i$  to  $v$ . A partial assignment,  $PA$ , is a set of assignments for all the variables in  $Y$ , where  $Y \subseteq X$ . A partial assignment is complete when  $Y = X$ .

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

**Definition 5.** Let  $var(PA) = \{x | (x, v) \in PA\}$ , the set of variables assigned values by a  $PA$ . Let  $val(PA) = \{v | (x, v) \in PA\}$ , the set of values assigned for  $var(PA)$ . A partial assignment  $PA$  satisfies a constraint  $c_i$ , when  $PA|_{var(PA) \cap S_i} \in R_i|_{var(PA) \cap S_i}$ .

**Definition 6.** A complete assignment  $CA$  is a solution  $S$  for the *CSP* when  $CA$  satisfies all the constraints in  $C$ .

*Example 1.* Fig. 1 shows an example *CSP*. In this example, there are five variables  $\{x_1, x_2, x_3, x_4, x_5\}$  and four constraints. A sample solution for the example is  $\{(x_1, 0), (x_2, 1), (x_3, 4), (x_4, 3), (x_5, 0)\}$ .  $\diamond$

$$\begin{aligned}
X &= \{x_1, x_2, x_3, x_4, x_5\} \\
D &= \{\{0, 1, 2\}, \{0, 1\}, \{4, 5\}, \{0, 1, 3\}, \{0, 1, 2\}\} \\
C &= \{((x_1, x_2), \{(0, 1), (2, 0)\}), \\
&\quad (\{x_2, x_3\}, \{(0, 4), (1, 4)\}), \\
&\quad (\{x_3, x_4, x_5\}, \{(4, 3, 0), (4, 0, 2), (5, 3, 2)\}), \\
&\quad (\{x_1, x_4\}, \{(0, 0), (0, 1), (1, 1), (0, 3)\})\}
\end{aligned}$$

**Fig. 1.** An example constraint satisfaction problem.

Given a *CSP* instance, the set of all complete assignments  $CAS \equiv D_1 \times D_2 \times \dots \times D_n$ . The number of possible complete assignments is  $|CAS|$ . Let  $SOL$  denote the set of all solutions of the *CSP*. Clearly,  $SOL \subseteq CAS$ .

Several real-world problems can be modelled as CSP instances, including: configuration, scheduling, rostering, and sequencing. In this report we focus on configuration problems modelled as CSP instances and present techniques for efficient interactive configuration.

**Definition 7.** *A customizable product specification is given by the available options in the product and the rules in which the choices for those options can be selected. Given a customizable product specification, the configuration problem CP is defined as selecting a choice for each of the available options, such that all the rules are satisfied.*

In this report, we only consider the static configuration problem, in which the number of available options and the rules remain the same throughout the configuration process. In case of dynamic configuration problems [9], some of the choices for the available options might add new options and rules to the configuration problem. Methods for static configuration problem can be easily extended to dynamic configuration problems [10].

*Example 2.* 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. Fig. 2 shows a T-shirt configuration problem as a CSP instance. The configuration problem 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  $C = \{c_1, c_2\}$ , where  $c_1 \equiv (x_3 = MIB) \Rightarrow (x_1 = black)$  and  $c_2 \equiv (x_3 = STW) \Rightarrow (x_2 \neq small)$ . There are  $|D_1||D_2||D_3| = 24$  possible CAs. Eleven of these assignments are valid configurations and they form the corresponding SOL shown in Fig. 3.  $\diamond$

**Definition 8.** *A configurator is the tool which helps the user in selecting his preferred product, which needs to be valid according to a given product specification. An interactive configurator IC is a configurator which interacts with the user as and when a choice is made by the user. After each and every choice selection by the user the IC shows a list of unselected options and the valid choices for each of them.*

The IC only shows the list of valid choices to the user. This prevents the user from selecting a choice, which along with the previous choices made by the user, if any, will result in no valid product according to the specification. The configurator, hence, automatically hides the invalid choices from the user. The *hidden choices* will still be visible to the user, but with a tag that they are inconsistent with the current PA. When the current PA is extended by the user, some of the choices might be implied for consistency. Such choices are automatically

$$\begin{aligned}
X &= \{x_1, x_2, x_3\} \\
D &= \{\{black, white, red, blue\}, \{small, medium, large\}, \{MIB, STW\}\} \\
C &= \{\{x_1, x_3\}, \{(black, MIB), (black, STW), (white, STW), (red, STW), (blue, STW)\}\} \\
&\quad \{\{x_2, x_3\}, \{(small, MIB), (medium, MIB), (medium, STW), \\
&\quad\quad (large, MIB), (large, STW)\}\}
\end{aligned}$$

**Fig. 2.** The T-shirt configuration problem as a CSP instance.

$$\begin{array}{lll}
(black, small, MIB) & (black, large, STW) & (red, large, STW) \\
(black, medium, MIB) & (white, medium, STW) & (blue, medium, STW) \\
(black, medium, STW) & (white, large, STW) & (blue, large, STW) \\
(black, large, MIB) & (red, medium, STW) &
\end{array}$$

**Fig. 3.** Solution space SOL for the T-shirt example.

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

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

1. Display
2. Propagate
3. Explain

**Definition 9.** *Display is the function, which given a CSP and a corresponding SOL', SOL'  $\subseteq$  SOL, lists X, the options in CSP and  $CD_i$ , the available valid choices, for each option  $x_i \in X$ , where  $CD_i = \{v | (x_i, v) \in S, S \in SOL'\}$ .*

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

**Definition 10.** *Propagate is the function, which given a CSP, a corresponding SOL', and  $(x_i, v)$ , where  $v \in CD_i$ , restricts SOL' to  $\{S | (x_i, v) \in S, S \in SOL'\}$ .*

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

**Definition 11.** *Let  $(x_{ih}, v_{ih})$  be an implied or hidden choice. Explain( $x_{ih}, v_{ih}$ ) is the process of generating, E, a set of one or more selections made by the user, which implies or hides  $(x_{ih}, v_{ih})$ . The E is called as an explanation for  $(x_{ih}, v_{ih})$ .*

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

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

**Definition 12.** *The cost of an explanation is  $\text{Cost}(\mathbf{E}) = \sum_{(x_i, v) \in \mathbf{E}} P(x_i, v)$ . An explanation  $\mathbf{E}$  is minimum, if there does not exist an explanation  $\mathbf{E}'$  for  $(x_{ih}, v_{ih})$ , such that  $\text{Cost}(\mathbf{E}') < \text{Cost}(\mathbf{E})$ .*

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

### 3 An algorithm for Iterative Configuration

Based on the definitions in the previous section, the algorithm for interactive configuration is given in Fig. 4. The `COMPILE` function will return a data-structure encoding all the valid solutions for the input configuration problem. In our case, the data-structure will be a BDD.

```

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

```

**Fig. 4.** The Interactive Configurator Algorithm.

## 4 Representing Solution Space using Binary Decision Diagrams

In this section, we will describe reduced ordered Binary Decision Diagrams (BDD) and show how it can be used to represent a solution space.

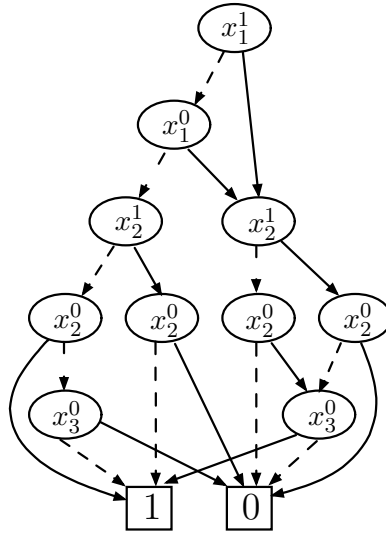
A BDD [11] is a rooted directed acyclic graph with two terminal nodes marked 1 and 0, respectively. All the non-terminal nodes, including the root, will be associated with a Boolean variable. Each non-terminal node will have two outgoing edges: *low* and *high*. All the non-terminal nodes will be ordered based on a linear variable order. A BDD is reduced such that no two nodes in the BDD have the same low and high successors, and no node has the same low and high successors. The variable ordering and the reductions make BDDs canonical. BDDs can be used to represent Boolean functions. Given any Boolean function, the corresponding BDD can be obtained by standard composition functions on BDDs representing atomic elements of the Boolean function. Given an assignment to the Boolean variables in the function, there exists a unique path from the root node to one of the terminal nodes, defined by recursively following the high edge, when the associated variable is assigned true, and the low edge, when the associated variable is assigned false. If the path leads to the terminal node 1, then the assignment is a solution, otherwise not. Given a BDD, any path from the root to the terminal node 1 represents one or more solutions. Any variable whose node is not present in such a path is a don't care variable in the solution.

Although the size of a BDD can be exponential in the worst case, reductions make the BDDs typically small for many practical functions. Due to this BDDs have been successfully used for representing solutions sets in several research areas, including: hardware verification [12, 13], constraint satisfaction [14, 15], planning [16], and configuration [4, 5]. The size of the BDDs are very sensitive to the used variable ordering. Finding the best variable ordering is itself a co-NP-complete problem. A good rule of thumb is to pick a variable order in which highly related variables are close in the ordering. Further details on BDDs can be obtained from [3, 17, 18].

When solution space of a non-Boolean function needs to be represented by a BDD, each non-Boolean variable  $x_i$  with domain  $D_i$  will be represented by  $l_i$  Boolean variables, where  $l_i = \lceil \lg |D_i| \rceil$ . A BDD corresponding to any rule(function) can be obtained by the composition function on BDDs representing the atomic elements of the rule. In case of CSP, the conjunction of the rules represent the solution space, and hence the conjunction function when applied to the BDDs obtained for all the rules in the CSP will give a single/monolithic BDD, which will represent the solution space of the CSP. As  $l_i$  Boolean variables could represent  $2^{l_i}$  values, in cases where  $|D_i| < 2^{l_i}$ , additional rules need to be added to maintain domain integrity. The following example shows how the solution space for the T-shirt example can be represented by using a BDD.

*Example 3.* The BDD representing the solution space of the T-shirt example 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,

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  $(x_i^{l_i-1}, \dots, x_i^1, x_i^0)$ , 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*), respectively. In this path the variable  $x_3$  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 one or more 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  $x_i$ .

#### 4.1 BDD operations

Let  $b$  be a BDD, and  $Var(b)$  denote the set of variables in  $b$ . Unambiguously, the notation  $b$  might also be used to denote the Boolean function represented by the BDD  $b$ . The following standard BDD operations will be used to implement a BDD-based configurator.

1.  $\text{Restrict}(b, x_i=v)$ : Given a BDD  $b$  and an assignment  $(x_i=v)$  where  $x_i$  is a Boolean variable and  $v$  is a Boolean value, the restrict operation will return a BDD  $b'$ , which will represent a function obtained by substituting each occurrence of  $x_i$  in  $b$  by  $v$ . The complexity of restrict is linear in the size of the given BDD. In our case, when we restrict a BDD by a non-Boolean variable assignment  $(y_i=yval)$ , the necessary effect will be achieved by a sequence of corresponding restrict operations on the Boolean variables that encode the non-Boolean variable  $y$ .
2.  $\text{Exist}(b, x_i)$ : The exist function will existentially quantify  $x_i$  from  $b$  and return a BDD  $b'$ , where  $b' = \text{Restrict}(b, x_i = 0) \vee \text{Restrict}(b, x_i = 1)$ . As the restrict operation is linear and the OR-operation ( $\vee$ ) is quadratic, the complexity of exist is quadratic. Usually existential quantification of a variable will result in a smaller BDD. When a non-Boolean variable or a set of non-Boolean variables are given as second argument, all the corresponding Boolean variables will be existentially quantified out.
3.  $\text{Conjoin}(b_1, b_2)$ : Conjoin operation will give the result of AND operation between  $b_1$  and  $b_2$ . This also has quadratic complexity.
4.  $\text{Proj}(b_1, b_2)$ :  $\text{Proj}(b_1, b_2) = \text{Exist}(\text{Conjoin}(b_1, b_2), \text{Var}(b_2) \setminus \text{Var}(b_1))$ . The complexity of projection depends upon the number of variables existentially quantified.
5.  $(b_1=b_2)$ : Due to canonicity property of BDDs, equality testing could be done in constant time.

Using the above BDD-operations the three operations required for interactive configuration - display, propagate, and explain - can be implemented. Display function can be implemented by an algorithm that traverses the paths from the root node to the 1-terminal node. Such an algorithm with polynomial complexity is explained in [19]. Propagate function can be implemented by a sequence of calls to the restrict function. In Section 6, algorithms for minimum explanation generation will be presented.

## 5 Tree Decomposition for CSP Compilation

Tree decomposition techniques [20, 6, 21] for CSPs can be used to obtain a compilation scheme with lesser storage requirement than the compilation schemes without decompositions. The following definitions [22] will be useful in discussing the tree decomposition techniques for CSPs.

**Definition 13.** *The constraint graph  $(V, E)$  of a given CSP will contain a node for each constraint in the CSP and an edge between two nodes if their corresponding constraints share at least one variable in their scopes. Each edge will be labelled by the variables that are shared by the scope of the corresponding constraints.*

**Definition 14.** *A subset of the edges,  $(E' \subseteq E)$ , in a constraint graph are said to satisfy the connectedness property, if for any variable  $v$  shared by two*

constraints, there exists a path between the corresponding nodes in the constraint graph, using only the edges in  $E'$  with  $v$  in their labels.

**Definition 15.** A join graph of a constraint graph contains all the nodes in the constraint graph, but the set of edges in the join graph is a subset of the edges in the constraint graph such that the connectedness property is satisfied. If the join graph does not have any cycles, then it is called a join tree.

Given a constraint graph of a CSP, it has a join tree if its maximum spanning tree, when the edges are weighted by the number of shared variables, satisfies the connectedness property [22]. Several tree decomposition techniques [20, 6, 21] have been proposed to convert a CSP into another one, CSP', such that CSP' has a join tree.

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

**Definition 16.** A constraint of a CSP is said to be minimal when all the solutions of the constraint can be extended to a solution for the CSP.

A CSP with a join tree is called acyclic CSP. Acyclic CSPs can be efficiently solved. The nice property of an acyclic CSP is that, mutual projections between constraints adjacent in its join tree, makes all the constraints minimal. Minimality of constraints is enough to obtain the efficient functions required for interactive configuration. Since each constraint is minimal, a valid choice for a variable in a constraint will also be a valid choice for the variable in the entire CSP, and the display function will use this to efficiently display valid choices. Given an assignment, the propagate function just needs to reduce all the constraints in which the variable is present, and propagate the effect to other constraints through projections. Hence, instead of compiling the conjunction of all the constraints in a CSP into a single/monolithic BDD, the tree decomposition based compilation scheme just requires a BDD for each constraint in the corresponding CSP'. We call the BDDs representing the constraints of CSP' as the *tree-of-BDDs*.

Since the product/configuration specifications have a hierarchical description, they might be amenable for CSP decomposition techniques. Since the sum of sizes of the BDDs in a tree-of-BDDs is potentially smaller than the size of the corresponding monolithic BDD, decomposition schemes might lead to significant space savings in CSP compilations.

We will use the hinge tree decomposition algorithm for CSP introduced in [6] for decomposing the configuration problems (CP). In the following subsections we explain the hinge decomposition algorithm and also give some additional details on the implementation of the display and propagate functions on a tree-of-BDDs.

## 5.1 Hinge Tree Decomposition

The hinge decomposition algorithm is given in Fig. 6. The following definition will be used in the algorithm.

**Definition 17.** [6] Let  $(V, E)$  be a pair of the set of variables and the set of scopes of constraints, both corresponding to a CSP. Let  $H \subseteq E$ , and let  $F \subseteq E - H$ .  $F$  is called connected with respect to  $H$  if, for any two scopes  $e, f \in F$ , there exists a sequence  $e_1, \dots, e_n$  of scopes in  $F$  such that (i)  $e_1 = e$ ; (ii) for  $i = 1, \dots, n - 1$ ,  $e_i \cap e_{i+1}$  is not contained in  $\bigcup H$ ; and (iii)  $e_n = f$ . The maximal connected subsets of  $E - H$  with respect to  $H$  are called the connected components of  $E - H$  with respect to  $H$ .

The algorithm takes as input a pair  $(V, E)$ , where  $V$  is the set of variables in a CSP and  $E$  is the set of scopes of the constraints in the CSP. The output of the algorithm is  $\text{CSP}'$ , which is acyclic, and a join tree of  $\text{CSP}'$ . The output is stored in  $N_{i+1}$  and  $A_{i+1}$ , where  $i$  is the iteration in which the algorithm stops. Each element in  $N_{i+1}$  is a set of some of the scopes of the constraints in the CSP. Each scope points to a unique constraint in the input CSP. Each element of  $N_{i+1}$  defines a cluster formed by the original constraints that correspond to the scopes in the element. Conjunction of the original constraints that correspond to an element in  $N_{i+1}$  form a constraint in  $\text{CSP}'$ . Each element in  $A_{i+1}$  is a pair of two elements in  $N_{i+1}$ , which defines an arc/edge between them in the output join tree. The elements in  $A_{i+1}$  define a join tree for  $\text{CSP}'$ . The algorithm runs in polynomial time. The proof of the correctness of this algorithm and additional details are given in the original paper [6].

HINGEDECOMPOSITION( $V, E$ )

- 1 Mark each scope in  $E$  as *unused*. Set  $i:=0, N_0:=\{E\}$  and  $A_0:=\emptyset$ , and mark the node  $E$  in  $N_0$  as *non-minimal*.
- 2 If all nodes of  $N_i$  are marked *minimal*, then set  $T:=(N_i, A_i)$  and stop. Else, choose a *non-minimal* node  $F$  in  $N_i$ .
- 3 If all scopes in  $F$  are marked *used*, then mark  $F$  as *minimal* and return to Step 2. Else, choose an *unused* scope  $e \in F$  and mark  $e$  as *used*.
- 4 Let  $\Gamma:=\{G \cup \{e\} \mid G \text{ is a connected component of } F - \{e\} \text{ with respect to } e\}$ , and let  $\gamma : F \rightarrow \Gamma$  be any function such that for all  $f \in F$ ,  $f \in \gamma(f)$ . If  $|\Gamma|=1$ , then return to Step 3.
- 5 Set,  
 $N_{i+1}:= (N_i - \{F\}) \cup \Gamma$ ,  
 $A_{i+1}:= (A_i - \{(\{F, F'\}, f) \mid (\{F, F'\}, f) \in A_i\})$   
 $\cup \{(\{\{\gamma(f), F'\}, f)\} \mid (\{F, F'\}, f) \in A_i\}$   
 $\cup \{(\{\{\gamma(f), \gamma(e)'\}, e)\} \mid f \in F, \gamma(g) \neq \gamma(e)\}$ ,  
and mark all the new nodes added to  $N_{i+1}$  as *non-minimal*.
- 6 Increment  $i$  and return to Step 2.

**Fig. 6.** The Hinge Tree Decomposition Algorithm for CSPs[6].

## 5.2 Configuration Functions on Tree-of-BDDs

Given a CP, we obtain an equivalent acyclic CP' using the hinge decomposition algorithm. Then, we compile each constraint in CP' into a BDD. The resulting set of BDDs along with the join tree obtained from  $A_{i+1}$  form the tree-of-BDDs. The tree-of-BDDs is made minimal by imposing arc-consistency by projections. Due to acyclicity of the CP', this could be done efficiently. Now, we define some data-structures which will be used for implementing the display and propagate functions, *DisplayDe* and *PropagateDe*, on a tree-of-BDDs.

1. *TreeNodes*(TN): A list of nodes in the join tree. There will be a node for each constraint in CP'. Each node will be represented by the BDD of the corresponding constraint.
2. *TreeNodeVars*(TV): A list of set of variables. One set for the variables that occur in each node of *TreeNodes*.
3. *AdjList* (AL): A list of adjacency list for each node in *TreeNodes*. Adjacency list of a node  $n$  is the list of nodes in *TreeNodes* that are adjacent to  $n$  in the join tree.
4. *VarNodeList*(VNL): A list of list of nodes, where each list of nodes corresponds to a variable, and lists the nodes in which the variable occurs.
5. *ModifiedQueue*(MQ): A queue to store the nodes whose corresponding BDDs were changed due to a propagation. The queue is maintained, so that the changes are propagated to the neighbours of the corresponding node.
6. *Redundant*(R): A list of Boolean values, one corresponding to each node in the *TreeNodes*. Since a variable could be present in more than one node of the tree, and the minimality of each node is maintained, *display* procedure on one among the nodes in which a variable is present is enough to obtain the current valid choices for the variable. The values in  $R$  are such that calling the display function on the nodes with  $R$  value set false is enough to obtain the remaining valid choices for all the variables in the CP. The values in  $R$  are populated by a heuristic which tries to maximize the number of nodes whose  $R$  value is set true, while covering all the variables. Experiments showed that several nodes could be redundant.
7. *ValidChoices*(VC): A set of pairs. Each pair is made up of a variable and its corresponding set of valid choices consistent with a PA.

The *PropagateDe* function for CP represented by a tree-of-BDDs is listed in Fig. 7. This is equivalent to the *Propagate* function in the monolithic BDD-based scheme. The *PropagateDe* function takes an assignment  $(x_i, v)$  as input, and makes the tree-of-BDDs consistent with the assignment. Due to acyclicity of CP', once a node is changed during an assignment, it will not change again due to projections on it by its neighbours. Hence, the propagation is one way, and the complexity of the function is polynomial.

The *DisplayDe* function for CP represented by a tree-of-BDDs is listed in Fig. 8. This is equivalent to the *Display* function in the monolithic BDD-based scheme. The *DisplayDe* function calls the *Display* function on all non-redundant nodes in the tree-of-BDDs, and prints the valid choices for all the variables in the CP. The complexity of *DisplayDe* is also polynomial.

```

PROPAGATEDE( $AL, VNL, (x_i, v)$ )
1   $MQ.Clear()$ 
2   $\forall N \in VNL[x_i]$ 
3     $N := Propagate(N, (x_i, v))$ 
4     $MQ.Push(N)$ 
5  while  $MQ.NotEmpty()$ 
6     $CURR := MQ.Pop()$ 
7     $\forall N \in AL[CURR]$ 
8       $N' := Proj(N, CURR)$ 
9      if  $(N' \neq N)$ 
10        $N := N'$ 
11        $MQ.Push(N)$ 
12  return

```

**Fig. 7.** The PropagateDe Algorithm for assignment propagation in the tree-of-BDDs scheme.

```

DISPLAYDE( $TN, TV, R$ )
1   $VD := \{\}$ 
2   $\forall N \in TN$ 
3    if  $(R[N] = \text{false})$ 
4       $VD := VD \cup Display(N)_{TV[N]}$ 
5  print  $VD$ 
6  return

```

**Fig. 8.** The DisplayDe for displaying valid choices in the tree-of-BDDs scheme.

## 6 Algorithms for Minimum Explanations

Efficient explanations have been an interesting topic of research in both AI and CP fields [23–25, 7]. In this section we present two algorithms for generating explanations for BDD-based interactive configurators. The first algorithm is for the monolithic-BDD based scheme and the second one for the tree-of-BDDs scheme. Both the algorithms generate minimum explanations with respect to a cost function. The first subsection presents a linear algorithm for minimum explanations in the monolithic-case. The explanation algorithm for the tree-of-BDDs scheme subsequently follows.

### 6.1 A Linear Algorithm for Minimum Explanations in the Monolithic-BDD Scheme

```
EXPLAIN( $b, PA, (x_{ih}, v_{ih})$ )
1   $b := b_{|(x_{ih}, v_{ih})}$ 
2   $NQ := \text{TopologicalSort}(b)$ 
3  Initialize( $b$ )
4  while  $NQ.\text{NotEmpty}()$ 
5       $n := NQ.\text{Pop}()$ 
6      Relax( $n$ )
7   $E := \text{'Explanation corresponding to the path to 1-terminal'}$ 
8  return  $E$ 
```

**Fig. 9.** The Explain algorithm for the monolithic-BDD scheme.

Let the *Explain* function be called with  $(x_{ih}, v_{ih})$  as argument. Let  $b$  be the BDD representing **SOL**. An explanation would correspond to a path from the root node of the BDD  $b$  to the 1-terminal node. Such a path must contain the assignment  $(x_{ih}, v_{ih})$ , and the path is said to violate an assignment  $(x_i, v) \in PA$ , if  $(x_i, v)$  is not encoded by the path. A minimum explanation then would correspond to such a path in which the sum of the costs of the violated assignments in **PA** is minimum. A minimum cost path could be obtained using a variant of a standard shortest path algorithm for directed acyclic graphs [26].

For simplicity the cost for violating each user assignments is assumed to be one. The general case is discussed at the end of this subsection, and could be easily obtained from the simpler one presented here. We also assume that all the Boolean variables representing a variable in a **CSP** are placed consequently in the variable ordering of the BDDs. This is a meaningful assumption, as such Boolean variables are highly related, and it is advisable to keep them together in the variable ordering for minimizing the size of the BDDs. Let  $n$  be a node in the BDD  $b$ ,  $\text{Var}(n)$  denote the Boolean variable  $x_i^k$  associated with the node, and

```

RELAX(n)
1  if (n.is_added=true  $\vee$  AV[Var(n)]=d)
2    leftcost := 0, rightcost := 0
3  else if (AV[Var(n)] $\neq$  d  $\wedge$  AV[Var(n)]=true)
4    leftcost := 1, rightcost := 0
5  else
6    leftcost := 0, rightcost:=1
7  left:=LOWCHILD(n), right:=HIGHCHILD(n)
8  if ( (left.cost > n.cost+leftcost)  $\vee$ 
9      (left.cost=n.cost+leftcost  $\wedge$  left.is_added=false) )
10   left.cost := n.cost+leftcost
11   left.parent := n
12   if (leftcost=1  $\wedge$  CSPVar(left)=CSPVar(n))
13     left.is_added := true
14   else if (leftcost=0  $\wedge$  CSPVar(left)=CSPVar(n))
15     left.is_added := n.is_added
16   else
17     left.is_added := false
18  if ( (right.cost > n.cost+rightcost)  $\vee$ 
19      (right.cost=n.cost+rightcost  $\wedge$  right.is_added=false) )
20   right.cost:=n.cost+rightcost
21   right.parent := n
22   if (rightcost=1  $\wedge$  CSPVar(right)=CSPVar(n))
23     right.is_added := true
24   else if (rightcost=0  $\wedge$  CSPVar(right)=CSPVar(n))
25     right.is_added := n.is_added
26   else
27     right.is_added := false
28  return

```

**Fig. 10.** The Relax procedure used by the Explain algorithm.

$\text{CSPVar}(n)$  denote the CSP variable  $x_i$  corresponding to  $\text{Var}(n)$ . The CSPVar of terminal nodes will be a null value. If the variable  $x_i^k$  is assigned true (false) in PA, then the high edge of node  $n$  will have a cost one (zero), and the low edge cost zero (one). If  $x_i^k$  is unassigned in PA, then both of its outgoing edges will have cost zero. Given a path from the root node to the 1-terminal node, the cost for violating a variable  $x_i$  needs to be counted only once. But, when two Boolean variables corresponding to  $x_i$  differs from the corresponding values assigned in PA, then the cost will be counted twice in a path. To prevent this, we use a flag, *is\_added*, associated with each node in the BDD. Once a cost is added for violating a variable  $x_i$ , then *is\_added* flag of the subsequent nodes will be set to *true*, until the path moves into a node belonging to a variable other than  $x_i$ . As in the classical shortest path algorithms, we also have two more values associated with each node in the BDD: *cost* and *parent*.

Now, the minimum explanation problem resembles closely to the shortest path problem in a directed acyclic graph. Such an algorithm is presented in Fig. 9. The algorithm uses *Relax*, an algorithm listed in Fig. 10. Given a node  $n$ , the relax procedure checks whether it is cheaper to reach its child nodes through  $n$  or the existing paths to the child nodes are itself cheaper. If either of the child nodes could be reached in lesser cost through  $n$ , the values of the corresponding child node will be changed accordingly. The conjunction in line 9 of the *Relax* procedure is necessary, because if the *left.is\_added* is set to *false* and both the costs are the same, the current path to *left* could potentially be costlier than the path through the corresponding  $n$ . AssignedVal (AV) is an array containing a value for all the Boolean variables in the BDD  $b$ . The values in AV will be true (false) if the corresponding variable is assigned true (false) in PA. Otherwise  $d$ , a don't care value. The *TopologicalSort* is a function, which will return a topologically sorted ordering of the nodes in an input BDD. In a topologically sorted order, if a node  $a$  appears before node  $b$ , then there will not be any path from  $b$  to  $a$  in the input BDD. The *Initialize* function will assign an infinite cost to all the nodes in the BDD. For the root node, the initialize function will assign a zero cost, a null parent, and set *is\_added* flag to false.

Since each node in a BDD has exactly two outgoing edges, the complexity of topological sorting is linear in the number of nodes in the BDD. The relax procedure is also called linear number of times. Hence, the complexity of the explanation algorithm is linear in the number of nodes. In the general case, when the cost need not be one and may be any positive value, then two paths might have to be remembered by each node. When the relax procedue is called for a node, a path to one of its child nodes with *is\_added* flag set to *true* may be costlier than another path with the flag set *false*. But still, the cheaper path with flag set *false* could potentially violate the PA in the consecutive nodes belonging to the same CSPVar. Until a CSPVar boundary is crossed, or the difference between the cost of two such paths is larger than the cost of the corresponding assignment, both the paths have to be remembered. To handle this problem two paths, one with *is\_added* flag set *false*, and another with *is\_added* set *true*, needs

to be remembered. When the paths cross a CSPVar boundary, only the best one among them will be selected. Clearly, still the algorithm would remain linear.

This algorithm was inspired by a similar algorithm discussed in [14]. However, the problem in [14] was made easier by adding an additional Boolean variable for each dynamic constraint added to a BDD. The additional variables will be used as switches to turn on or off the corresponding dynamic constraints. The additional variables will significantly increase the size of the BDD. In our case, the dynamic constraints are unary, and we assume that the Boolean variables corresponding to a CSPVar are placed consequently in the variable order. This requires the need for a single flag to ensure a minimum explanation.

## 6.2 A Minimum Explanation Algorithm for the Tree-of-BDDs Scheme

The explanation algorithm for the monolithic case can also be used to obtain an explanation generator for the tree-of-BDDs case. This relies on two facts:

1. In configuration problems, just assigning values for less than half of the variables in the CP, will imply a value for the rest of the variables and result in a CA. We have experimentally observed this during our previous experiments on several configuration instances [5].
2. When all the variables other than those in the PA are existentially quantified out from the BDD representing  $\text{SOL}_{|(x_{ih}, v_{ih})}$ , the explanation algorithm of the previous subsection will still be able to produce a minimum explanation. Hence, the unnecessary variables (those not in the PA) can be abstracted away using existential quantifications.

Given a tree-of-BDDs representing  $\text{SOL}_{|(x_{ih}, v_{ih})}$ , adjacent BDDs in the tree-of-BDDs can be conjoined together until a single BDD is obtained, which can be given as input to the explanation algorithm of the previous subsection to obtain a minimum explanation. But the problem with this approach is that the BDD obtained at the end will be as large as the corresponding BDD in the monolithic case, and hence, we lose the benefit of using tree-decomposition for reducing space usage. But using the above listed facts, unnecessary variables could be abstracted away by existential quantification, as and when they appear in only one BDD of the tree-of-BDDs. The sketch of the explanation procedure based on this technique is listed in Fig. 11. For simplicity, the changes made to the other datastructures, like TV and VNL, are not shown in the algorithm.

## 7 Efficient Full Interchangeability Detection for CSP Reformulation

In this section, we present a method to detect full interchangeable (FI) values of a CSP in both the monolithic-BDD and the tree-of-BDDs scheme.

```

EXPLAINDE( $PA, (x_{ih}, v_{ih})$ )
1   $TN' := SOL_{|(x_{ih}, v_{ih})}$ 
2  while ( $TN'.size() \neq 1$ )
3     $b_1 := TN'.Pop()$ 
4     $b_2 := TN'.Pop()$ 
5     $b_{12} := Conjoin(b_1, b_2)$ 
6     $QuantifyUnnecessaryVars(b_{12})$ 
7     $TN'.Push(b_{12})$ 
8   $E := Explain(TN'.Pop(), PA, (x_{ih}, v_{ih}))$ 
9  return  $E$ 

```

**Fig. 11.** Sketch of the ExplainDe algorithm for the tree-of-BDDs scheme

**Definition 18.** [27] A value  $\mathbf{a}$  for a variable  $\mathbf{x}_i$  is fully interchangeable with a value  $\mathbf{b}$  if and only if every solution in which  $\mathbf{x}_i = \mathbf{a}$  remains a solution when  $\mathbf{b}$  is substituted for  $\mathbf{a}$  and vice-versa.

Using a monolithic-BDD representing  $SOL$ , fully interchangeable values can be easily determined. Two values of a variable  $x_i$  are fully interchangeable, when all the paths from the root node of  $SOL$  to its 1-terminal node, containing either one of the values, has a *similar* path containing the other value. The later path shares the first path in nodes corresponding to all variables other than  $x_i$ . Based on this observation, a polynomial procedure similar to the *Display* function, which traverses all solution paths in a BDD can be used to find full interchangeable values. But, we describe here a simpler algorithm using existential quantifications.

**Theorem 1.** A value  $\mathbf{a}$  for a variable  $\mathbf{x}_i$  is fully interchangeable with a value  $\mathbf{b}$  if and only if  $Exist(SOL_{|(x_i, \mathbf{a})}, x_i) = Exist(SOL_{|(x_i, \mathbf{b})}, x_i)$  and vice-versa.

Proof of this theorem is simple, based on the definition of restrict function and existential quantification.

Given a variable  $x_i$  with domain  $\{a_1, \dots, a_n\}$ , the BDDs corresponding to  $\forall_{k=1}^n Exist(SOL_{|(x_i, a_k)}, x_i)$  can be used to find interchangeable value sets of the variable  $x_i$ . After that, the corresponding CSP could be reformulated by choosing a representative value for each interchangeable value sets. The advantage of such reformulations is that with the reduction in the domain size of a variable, the problem gets simpler. Reformulations could be made transparent to the user by appropriately changing the *Display* function. This method could be extended to tree-of-BDDs scheme as follows.

**Definition 19.** [27] A value  $\mathbf{a}$  for a variable  $\mathbf{x}_i$  is neighbourhood interchangeable (NI) with a value  $\mathbf{b}$  if and only if for every constraint on  $\mathbf{x}_i$ , the values compatible with  $\mathbf{x}_i = \mathbf{a}$  are exactly those compatible with  $\mathbf{x}_i = \mathbf{b}$ .

By the definitions of FI and NI,  $NI \Rightarrow FI$ , while  $FI \not\Rightarrow NI$ .

**Theorem 2.** *When all the constraints in a CSP are minimal,  $FI \Leftrightarrow NI$*

The proof of the above theorem is simple as any solution of a minimal constraint can be extended to a solution for the entire CSP.

Now, in case of the tree-of-BDDs scheme, arc-consistency between adjacent nodes in its join tree makes its constraints minimal. For a given variable  $x_i$ , the NI value sets can be deduced by using the algorithm, explained for finding FI value sets in the monolithic-BDD scheme, on all the nodes in which  $x_i$  is present. By the above theorem, the obtained NI value sets are equivalent to the FI value sets.

An example illustrating the benefit of FI value sets detection and reformulation is shown in Fig. 12. This example is taken from a *PC* configuration problem [28]. In this example, the variable *GraphicsCardId* has six values in its domain. Hence, it needs three Boolean variables in a BDD representing SOL. But after FI value sets detection and reformulation, the domain size reduces to two, and hence just a single Boolean variable is enough to encode the variable. This will reduce the size of the BDD. The extent of the space reduction depends on the position of the variable in the BDD variable ordering and the structure of the PC instance.

*GraphicsCardId* = {*Asus, Diamond, Creative, ATIRage, Matrox, ATIBulk*}  
 FI value sets for *GraphicsCardId* are {*Asus, Diamond, Creative, Matrox*}  
 and {*ATIRage, ATIBulk*}

Domain of *GraphicsCardId* in the reformulated model is {*Asus, ATIRage*}

**Fig. 12.** An example for FI value sets detection and reformulation

## 8 Experimental Results

Experiments are done by implementing the techniques presented so far, on top of CLab [29, 19], an open source interactive configurator based on the monolithic-BDD scheme. CLab does not have an explanation facility, and the linear minimum explanation algorithm was implemented in it. The tree-of-BDDs scheme is implemented as a tool called iCoDE (interactive Configurator with Decompositions and Explanations). The iCoDE source code, which includes CLab with explanations, is available at [30]. A Pentium Xeon machine with 4GB RAM and 1MB L2 Cache is used in the experiments. The default variable ordering, the order in which variables appear in the input file, is used.

Two benchmark configuration instances are used in the experiments: PC and Renault [7]. Both of them are available on the web [28]. PC instance is a personal computer configuration problem. Renault instance is a car configuration problem, and it was the only instance used in the experiments of [7]. Renault

instance is quite large, the input file size is around 23 Megabytes, and it has totally around 200,000 tuples in 113 constraints of it. The characteristics of the benchmarks are listed in Table 1. The number of variables and the number of constraints in each instance is listed.  $\sum |D_i|$  refers to the sum of the domain size of the variables in the instance. Arity of a constraint is the size of its scope. Max  $a$  refers to maximum arity of the constraints.  $\sum a_i$  refers to the sum of the constraint arities. |SOL| refers to the number of solutions in the instance.

**Table 1.** Configuration Benchmarks

Benchmark	Variables	$\sum  D_i $	Constraints	Max $a$	$\sum a_i$	SOL
PC	45	383	36	16	107	$1.19 \times 10^6$
Renault	99	402	113	10	588	$2.84 \times 10^{12}$

The compilation details are listed in Table 2. CLab and iCoDE refer to the monolithic-BDD and the tree-of-BDDs scheme, respectively. The column "Hinge(sec)" refers to the time taken by the Hinge CSP Decomposition algorithm. Cyclicity is the size of the maximum-sized cluster in the output of the hinge decomposition. Lower cyclicity is preferable and higher cyclicity means a large cluster size and it will require more space. As specified in [6], cyclicity is an invariant for a given instance. Shuffling the constraints sequence in an input instance will not change the cyclicity of the instance. TreeNodes refers to the number of constraints in the output of the hinge decomposition. The following columns list the time taken for compilation in both the schemes. In the larger instance, Renault, iCoDE takes only around 50% of the time taken by CLab. The iCoDE timings could be considerably improved by making the BDDs corresponding to the original constraints arc-consistent, before building the BDDs for each cluster in the join tree. In the present implementation, arc-consistency is done only after the BDDs for each cluster is obtained. The compilation timings include the time taken for finding FI values. Reformulating the instances, by removing FI values, resulted in decreasing the space requirement by around 10% in both the instances. In case of iCoDE, the instance will be reformulated by exploiting the fully interchangeable values. BDD-nodes for CLab, refers to the number of nodes in the monolithic-BDD obtained after compilation. In case of iCoDE, BDD-nodes refers to the total number of nodes in the tree-of-BDDs. As in any standard BDD package, there is a chance for two BDDs in the hinge tree to share some BDD-nodes, and such shared nodes will be counted only once. Hinge decomposition(iCoDE) for Renault instance results in 96% decrease in the number of BDD-nodes required. Peak BDD-nodes refers to the number of nodes required by the BuDDy package [31], the BDD package used in CLab (iCoDE), to finish the compilation process. Even if the final BDD is small, the intermediate BDDs required during the compilation process may be very large, and hence the interest in Peak BDD-nodes. For the Renault instance, iCoDE

uses only 80,000 BDD-nodes to finish the compilation process, which is 97% less than that used by CLab.

**Table 2.** Compilation Details

Benchmark	Hinge (sec)	Cyclicity	Tree Nodes	Compile (sec)		BDD-nodes		Peak BDD-nodes	
				CLab	iCoDE	CLab	iCoDE	CLab	iCoDE
PC	0.04	21	16	0.11	0.19	16494	4458	80K	40K
Renault	0.25	25	73	119	77	455796	17602	2.5M	80K

Response times of both the schemes are compared in Table 3. The values listed are obtained from 10,000 random interactions. Each response is the sum of the time taken for a call to the *Propagate* function and a subsequent call to the *Display* function. During the random interactions, the time taken for calls to *Explain* function was also measured, and the corresponding timings are listed in Table 4. ART and WRT refers to the average and worst response time, respectively. AET and WET refers to the average and worst time taken for minimum explanations, respectively. In case of Renault instance, iCoDE results in 200X speedup for ART and 20X speedup for WRT. In explanation results for Renault instance, even though the *ExplainDe* function is not linear like *Explain*, in average case it takes lesser time to generate an explanation. This is due to 96% decrease in the BDD-nodes value. The WET by iCoDE is around twice the time taken by CLab. We believe that WET of iCoDE could be improved well, as the quantification scheduling of *ExplainDe* is naive. Several non-naive approaches like [32], could be used to improve the WET for iCoDE. Efficient techniques for building BDDs [33] could also be used to reduce the WET of iCoDE. Also, the explanation function in CLab is a contribution of this work.

**Table 3.** Response time comparison

Benchmark	ART(sec)		WRT(sec)	
	CLab	iCoDE	CLab	iCoDE
PC	0.004	0.0001	0.050	0.006
Renault	0.070	0.0003	0.452	0.020

## 9 Related work

Only a very few published CSP compilation techniques use tree decomposition techniques. We are not aware of any work detecting *full* interchangeable values

**Table 4.** Explanations time comparison

Benchmark	AET (sec)		WET (sec)	
	CLab	iCoDE	CLab	iCoDE
PC	0.004	0.010	0.010	0.030
Renault	0.160	0.088	0.440	0.921

for problem reformulation. In [34] a compilation scheme was presented, but it did not use tree decomposition techniques to guarantee a polynomial response time. Also, it only exploited the neighbourhood interchangeability. In [7], an automaton based compilation scheme without decomposition was presented. It is similar to the monolithic-BDD scheme. In [8], a CSP compilation technique combining tree clustering [20] and cartesian product representation (CPR) was presented. That work did not have minimum explanations and full interchangeability detection. Also, the CPR is not as well studied as the BDD, for representing solution sets. In [35], Tree-Driven Automata, a CSP compilation technique combining automata and tree decomposition was presented. It did not focus on the functionalities, like explanations, required for interactive configuration. Also, it did not have any experimental results.

## 10 Conclusion

A decomposition scheme, tree-of-BDDs, for compiling models for interactive configurators was presented. The decomposition scheme results in a drastic reduction in space required for storing the compiled solutions. A linear algorithm for minimum explanations in the monolithic-BDD scheme was given. Using abstractions, the explanation algorithm was extended to work in the tree-of-BDDs scheme. Procedures for exploiting full interchangeable values was given. All the techniques presented here were experimentally evaluated as useful. Altogether, we believe that this work improves the state-of-the-art in configurators and CSP compilation techniques.

Future work include: experiments on BDD variable orderings, hybrid representations instead of just BDDs, using multi-valued decision diagrams instead of BDDs [36], efficient quantification scheduling for explanations in the tree-of-BDDs scheme, and comparing other tree decomposition techniques with hinge decomposition technique.

## References

1. Configit Software A/S. <http://www.configit-software.com> (online)
2. Array Technology. <http://www.array.dk/> (online)
3. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **8** (1986) 677–691

4. 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
5. 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: Proceedings of Workshop on CSP Techniques with Immediate Application, International Conference on Principles and Practice of Constraint Programming. (2004) 97–111
6. Gyssens, M., Jeavons, P.G., Cohen, D.A.: Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence* **66** (1994) 57–89
7. Amilhastre, J., Fargier, H., Marquis, P.: Consistency restoration and explanations in dynamic CSPs-application to configuration. *Artificial Intelligence* **1-2** (2002) 199–234 <ftp://ftp.irit.fr/pub/IRIT/RPDMP/Configuration/>.
8. Madsen, J.N.: Methods for interactive constraint satisfaction. Master’s thesis, Department of Computer Science, University of Copenhagen (2003)
9. Mittal, S., Falkenhainer, B.: Dynamic constraint satisfaction problems. In: Proceedings of the 8th National Conference on Artificial Intelligence, AAAI Press/The MIT Press (1989) 25–32
10. van der Meer, E.R., Andersen, H.R.: BDD-based recursive and conditional modular interactive product configuration. In: Proceedings of Workshop on CSP Techniques with Immediate Application, International Conference on Principles and Practice of Constraint Programming. (2004) 112–126
11. Bryant, R.E.: Symbolic manipulation of boolean functions using a graphical representation. In: DAC. (1985) 688–694
12. 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
13. 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
14. Bouquet, F., Jégou, P.: Using OBDDs to handle dynamic constraints. *Information Processing Letters* **62** (1997) 111–120
15. Lagoon, V., Stuckey, P.: Set domain propagation using ROBDDs. In: Proceedings of the 9th International Conference on Principles and Practices of Constraint Programming, LNCS, Springer-Verlag (2004) 347–361
16. Jensen, R.M., Bryant, R.E., Veloso, M.M.: SetA\*: An efficient BDD-based heuristic search algorithm. In: Proceedings of 18th National Conference on Artificial Intelligence (AAAI’02). (2002) 668–673
17. Meinel, C., Theobald, T.: Algorithms and Data Structures in VLSI Design. Springer (1998)
18. Wegener, I.: Branching Programs and Binary Decision Diagrams. Society for Industrial and Applied Mathematics (SIAM) (2000)
19. Jensen, R.M.: Clab 1.0 user manual. Technical report, IT University of Copenhagen (2004) ITU-TR-2004-46.
20. Dechter, R., Pearl, J.: Tree-clustering schemes for constraint-processing. *Artificial Intelligence* **38** (1989) 353–366
21. Gottlob, G., Leone, N., Scarcello, F.: A comparison of structural CSP decomposition methods. *Artificial Intelligence* **124** (2000) 243–282

22. Dechter, R.: Constraint Processing. Morgan Kaufmann (2003)
23. Junker, U.: QUICKXPLAIN: Conflict detection for arbitrary constraint propagation algorithms. In: IJCAI'01 Workshop on Modelling and Solving problems with constraints. (2001)
24. Jussien, N.: e-constraints: explanation-based constraint programming. In: CP01 Workshop on User-Interaction in Constraint Satisfaction. (2001)
25. Freuder, E.C., Likitvivatanavong, C., Moretti, M., Rossi, F., Wallace, R.J.: Explanations and optimization in preference-based configurators, LNCS 2627, Recent Advances in Constraints, Springer (2002) 58–71
26. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms (Second Edition). MIT Press/McGraw-Hill (2001)
27. Freuder, E.C.: Eliminating interchangeable values in constraint satisfaction problems. In: Proceedings of the 9th National Conference on Artificial Intelligence, AAAI Press / The MIT Press (1991) 227–233
28. CLib: Configuration benchmarks library. <http://www.itu.dk/doi/VeCoS/clib/> (online)
29. Jensen, R.M.: CLab: A C++ library for fast backtrack-free interactive product configuration. <http://www.itu.dk/people/rmj/clab/> (online)
30. Subbarayan, S.: iCoDE: An interactive Configurator with Decompositions and Explanations. <http://www.itu.dk/people/sathi/icode/> (online)
31. Lind-Nielsen, J.: BuDDy - A Binary Decision Diagram Package. <http://sourceforge.net/projects/buddy> (online)
32. Chauhan, P., Clarke, E.M., Jha, S., Kukula, J.H., Shiple, T.R., Veith, H., Wang, D.: Non-linear quantification scheduling in image computation. In: Proceedings of ICCAD. (2001) 293–299
33. Cabodi, G., Camurati, P., Quer, S.: Dynamic scheduling and clustering in symbolic image computation. In: Proceedings of DATE. (2002) 150–157
34. Weigel, R., Faltings, B.: Compiling constraint satisfaction problems. Artificial Intelligence **115** (1999) 257–287
35. Fargier, H., Vilarem, M.C.: Compiling CSPs into tree-driven automata for interactive solving. Constraints **9** (2004) 263–287
36. Kam, T., Villa, T., Brayton, R.K., Sangiovanni-Vincentelli, A.L.: Multi-valued decision diagrams: Theory and applications. Multiple-Valued Logic: An International Journal **4** (1998) 9–62