

BDD-based Recursive and Conditional Modular Interactive Product Configuration

Erik R. van der Meer and Henrik Reif Andersen

Department of Innovation, IT University of Copenhagen,
Rued Langgaards Vej 7, DK-2300 Copenhagen S, Denmark
{ervandermeer,hra}@itu.dk

Abstract. Interactive product configuration is a difficult problem in constraint programming. One of the reasons for this is that interactivity requires that the system always responds fast. One of the ways to deal with this requirement is to adopt a two-stage approach in which a product model is compiled off-line before user interaction takes place. One such approach uses reduced ordered Binary Decision Diagrams (BDDs) as the compilation target for the product model. Advantages of this representation are that it is often quite compact, that interactive configuration can take place in time linear of the size of the representation, and that this size is only dependent on the solution set (the semantics) and not on the description (the syntax) of the product model. In this paper, we extend this approach with the notion of modules, which are more loosely coupled parts of a configuration problem that often correspond with actual modules in the actual product. We allow for the combining of such modules by means of shared variables, for conditional instantiation of such modules, and for recursion between modules which leads to more natural descriptions of modular systems that could, in principle, be a priori unbounded in size.

1 Introduction

Interactive product configuration is often presented as an application in constraint programming [1, 2]. One of the main problems in implementing a good interactive product configurator is the requirement of responding sufficiently fast although the problem is NP-hard. Therefore, sometimes an off-line pre-computation phase is used to crack the problem so that the interactive phase can work in time polynomial in the size of the intermediate representation [3, 4]. One such approach is based on compilation to reduced ordered Binary Decision Diagrams (BDDs), which can represent many practical solution sets very compactly.

This is the approach we have chosen to extend. In this paper, we present the formal semantics of the pre-computation and interaction phases of the recursive and conditional modular interactive configuration problem. In a modular configuration problem, each module is a configuration problem by itself, but the modules can share variables, as long as the dependency graph for the modules

is a directed acyclic graph, and the run-time dependency graph for the module instances is a tree. In the conditional modular configuration problem, module instances in the tree exist or not depending on conditions in the parent module instance. In the recursive conditional modular configuration problem, the dependency graph for the modules is a directed graph, and the run-time dependency graph for the module instances is a tree of unbounded but finite size.

We achieve strong guarantees on user guidance, in that all configuration is backtrack-free, while the system is sound (no invalid/inconsistent solutions reachable), complete (all finite valid consistent solutions reachable), and terminable (finite valid consistent solution reachable at all times). This can be achieved in run-time linear in the number of modules. However, where in the non-modular configuration problem the run-time can be exponential in the size of the original problem, the modular configuration problem can take time exponential in the sizes of the original modules, or time exponential in the combined domain sizes of the shared variables. (If it did not, $P = NP$ could be answered affirmatively.) However, typical configuration is very efficient.

We have implemented the configuration system described above, and are currently using it to model a real world modular configuration problem. Thus far, we have studied a 2600 line model consisting of 28 modules, which compiles within a second on a 500 MHz Pentium III, and on which user interaction is handled without any noticeable delay. The model is not finished yet, but the BDDs seem to be growing gently as information is added. Thus far, the largest BDD found (an intermediate result during compilation) is smaller than 2000 nodes. Given the strong guarantees our system provides, we believe that we are achieving good results.

2 BDD-based Modular Interactive Configuration

In the modular configuration problem, a module is an entity with variables, domains, and constraints of its own. It can be instantiated by another module, with which it then shares the variables it exports, and it can instantiate other modules, with which it then shares the variables they export. In this way, the module's instances form a tree at run-time.

In programming language terms, a module can to some extent be thought of as a class, from which objects are instantiated. Also, the usual hierarchy building is possible. A car can be a module, with submodules like door, roof, and engine, and each of those can in turn have submodules.

2.1 Language

Our modular configuration language has these statements:

```

module M;
export x1, ..., xn;
import M as I if C;
define x : v1, ..., vn;
ensure C;

```

where x is a variable name, v a value name, M a module name, I an instance name, and C a constraint satisfying the following grammar:

$$\begin{array}{l}
C ::= (C) \\
\quad | !C \\
\quad | C_1 \text{ op } C_2 \text{ where } op \in \{ \&, |, -, <-> \} \\
\quad | x \text{ op } v \text{ where } op \in \{ <, =, >, <=, <>, >= \} \\
\quad | x_1 \text{ op } x_2 \text{ where } op \in \{ <, =, >, <=, <>, >= \}
\end{array}$$

Names are strings of alphanumerical characters, underscores and periods. The period is only used in names that refer to variables that are shared with imported modules.

An example of a single module: Here is a small t-shirt example, which consists of a single module:

```

module t_shirt;

define size : small, medium, large;
define color : red, blue, black, white;
define print : save_the_whale, men_in_black;

ensure print = save_the_whale -> size > small;
ensure print = men_in_black -> color = black;

```

The semantics of this will be intuitively clear. However, it is interesting to note that even though the example is trivially simple, not many people realize that when they want a small t-shirt, everything is decided.

An example of recursion: Next, we show an example that shows a toy application for recursion; that of configuring a USB tree:

```

module USB;

define type : unused, scanner, printer, camera, hub;

import scanner as scanner if type = scanner;
import printer as printer if type = printer;

```

```

import camera as camera if type = camera;

import USB as hub_connection_1 if type = hub;
import USB as hub_connection_2 if type = hub;
import USB as hub_connection_3 if type = hub;
import USB as hub_connection_4 if type = hub;

```

This shows the power of being able to use recursion in combination with conditional instantiation.

An example of potential conflict: Next, we show a short example showing a potential problem with the conditionality of import. Here is module A:

```

module A;

define x : 0, 1, 2;

import B as b if x = 1 | x = 2;

ensure x = 1 -> b.x <= b.y;
ensure x = 2 -> b.x < b.y;

```

And here is module B:

```

module B;

export x, y;
define x : 0, 1;
define y : 0, 1;

ensure x >= y;

```

The point here is that, while selection of $x = 1$ is fine, $x = 2$ leads to a conflict. Therefore, 2 should be removed from the domain of variable x . Please note that, since instantiation conditions can be arbitrarily large expressions, and since conflicts can arise from any combination of modules, this is not necessarily easy. However, our configurator handles this correctly while maintaining fast run-time response, stopping the user if and only if necessary.

An example of potential infinity: Next, we show a short example showing a potential problem with the recursiveness of import. Here is module A:

```

module A;

```

```

export x;
define x : 0, 1, 2;

import A if x = 1 | x = 2;

ensure x = 2 -> A.x = 2;

```

The point here is that, while selection of $x = 1$ is fine, $x = 2$ leads to the requirement to instantiate an infinite number of modules. Clearly, this needs to be avoided, again by removing 2 from the domain of x . Again, this situation can occur in more complicated ways, for example when different modules instantiate each other recursively. In fact, the same module can even occur multiple times in such a loop! However, our configurator handles this correctly while maintaining fast run-time response, again only stopping the user if and only if necessary.

Another example of potential infinity: Next, we show a short example showing a related potential problem with the recursiveness of import. Here is module A:

```

module A;

export x;
define x : 0, 1, 2, 3, 4;

import A as a1 if x = 1 | x = 3;
import A as a2 if x = 2 | x = 4;

ensure x = 1 -> a1.x <= 2;
ensure x = 2 -> a2.x <= 2;
ensure x = 3 -> a1.x >= 3;
ensure x = 4 -> a2.x >= 3;

```

The point here is that, while selection of $x = 0, 1$ or 2 is fine, $x = 3$ or 4 leads to trouble. The problem here is again one of instantiating an infinite number of modules, only now in lock step with user choices. That is, the configurator does not have to instantiate an infinite number of modules right away, but the user is doomed to keep on configuring for ever and ever. A configurator should avoid this, by removing the values 3 and 4 from the domain of x . Again, this problem can occur in more complicated forms. Still, our configurator handles this correctly while maintaining fast run-time response, stopping the user if and only if necessary.

2.2 Main Concepts of the Semantics

Compiling a single-module configuration problem is the process of determining the solution set of that module, that is the set of all combinations of values for the variables that satisfy the constraints. The user interaction process of single-module configuration is the process of beginning with that solution space, and reducing it step by step as the user assigns values to variables. During this process, we constantly keep the user informed about the remaining values that she can still choose for the still unassigned variables.

In modular configuration, this picture has to be extended. During modular configuration, there will be a tree of module instances, each of which has a solution space that we call the *local solution space*. Each of these will be stepwise reduced by the user, until their final configurations are found. Of course, it is necessary for the configurator to maintain global consistency over these local solution spaces. That is to say that each solution in a local solution space should be consistent with at least one solution in each of the neighbouring module instances' solution spaces. Consistent here means that the same variables (say x in child B, and $b.x$ in parent A) have the same values.

The *global solution space* is the product of all the local solution spaces. Unlike the local solution spaces, the global solution space can also be made larger by user selections. This happens when a user selection leads to the creation of a new instance, which introduces a new local solution space.

An instance is created when an instantiation condition in an import statement somewhere becomes true for all remaining solutions in the local solution space. In that case, it is known that for all configurations that can come out of the configuration process, the instance is going to be part of the global configuration.

There are three main things being done in the semantics. The first has to do with the propagation of variable names and domains, the second with the prevention of conflicts because of instantiation, and the third with the prevention of infinity problems.

Let us first consider the propagation of variable names and domains. Basically, when a variable is exported in a module, and another module imports that module, then the variable name and its domain are automatically added to the importing module. For example, if module B exports its variable x , then module A which imports module B (with instance name B1, say) automatically gets a variable B1. x . Since modules can import each other recursively, and since it is also possible for a module to export a variable it imported (say A exports B1. x), this propagation is formulated as a fixpoint operation.

Next, conflicts because of instantiation are avoided during the compilation phase: Whenever a module A imports a module B, the constraints that module B imposes on module A when it gets instantiated are already imposed implicitly on module A at compile time. This ensures that any local solutions in A that would instantiate B, but are not compatible with it, are removed from A's solution set.

Finally, the possibility of infinite regression at run-time is avoided at compile-time by a second fixpoint iteration. After the modules have been compiled sepa-

rately (to determine their local solution spaces), the compiler starts this fixpoint iteration to determine which of the local solutions of a module would allow that module to be at the root of a finite subtree in the instance tree. Only those solutions will make it to the final solution set. The fixpoint iteration starts with the local solutions for modules that do not instantiate any submodules, and then adds repeatedly those local solutions for modules that only instantiate submodules that have compatible solutions that have already been discovered in this fixpoint iteration.

It is important to note that our prototype configuration engine does not represent sets by explicit enumeration of their elements, but that it uses BDDs instead. This makes it a lot more efficient in practice. Furthermore, during the interaction phase, no changes are made in the BDDs, unless the user changes focus from one module to the other, and even then changes are only made to BDDs belonging to the modules along the path between these two. This is not readily apparent from the semantics, though.

2.3 Basic Definitions for the semantics

We now give the semantics of the modular configuration problem. These semantics only deal with user selections, not deselections, and they only contain equivalence as a comparison operator. We do not consider these to be serious drawbacks, because the meaning of deselection can be given in terms of a replay of the remaining selections, and the other comparison operators can easily be translated into a disjunctive subformula. The semantics are presented in this way in order to keep them simpler. However, our prototype engine implements everything directly.

We start the description of the semantics of a modular configuration problem with some preliminary definitions. First, we define \mathcal{M} , which is a sequence of modules.

$$\mathcal{M} = \langle M_1, \dots, M_n \rangle$$

Then there is the *addprefix* function, which takes an instance name and a sequence of variable names, and returns the sequence of variable names with the instance name (and a dot) prefixed to each of them.

$$\text{addprefix}(\text{instname}, \langle x_1, \dots, x_n \rangle) = \langle x'_1, \dots, x'_n \rangle$$

where

$$x'_i = \text{instname} \cdot \langle \cdot \rangle \cdot x_i$$

More important is the notion of projecting a tuple of values (representing a configuration of a single module) onto the interface with another module.

The *project* function here will take a tuple in which the values are from the domains of the variables in the sequence X , and will transform it into a tuple in which the values are from the domains of the variables in the sequence X' . All

variables in X' must occur in X , but the reverse does not have to be the case. Note the trick with the function i , which denotes the mapping from X to X' . It is defined in the second subequation, and used in the right hand side of the main equation.

$$\begin{aligned} \text{project}(X, X', (v_1, \dots, v_n)) &= (v_{i_1}, \dots, v_{i_m}) \\ \text{where} \\ \langle x_1, \dots, x_n \rangle &= X \\ \langle x_{i_1}, \dots, x_{i_m} \rangle &= X' \end{aligned}$$

Finally, there are the notions of projecting a set of tuples onto a set of shorter tuples, and of embedding a set of shorter tuples into a set of tuples. These operations are used to project solution sets in one module onto the interface to another, and then to embed them into the other solution domain.

Please note that `projectset` *projects* from X to X' , whereas `embedset` *embeds* into X from X' , and that X' should be the shorter sequence of variables of the two. In the case of `embedset`, we also need to pass in the entire domain D (the set of all valid and invalid configurations), since we need to know the full domains of the variables in X that do not occur in X' . S denotes a set of solutions, that is, a set of valid configurations within a module, and S' denotes such a set projected onto an interface.

$$\begin{aligned} \text{projectset}(X, X', S) &= \{\text{project}(X, X', V) \mid V \in S\} \\ \text{embedset}(X, X', D, S') &= \{V \mid V \in D, \text{project}(X, X', V) \in S'\} \end{aligned}$$

2.4 Compilation of Names

The first phase in the compilation of a modular configuration problem consists of the propagation of the definitions of exported variables from children to parents. Since modules can import each other recursively, this propagation is carried out as a fixpoint iteration.

The following functions obtain sequences of variable names (X), exported variable names (X^E), and the corresponding domains (D) from (sequences of) define and export statements. Note that the domain D is not a sequence of the domains of the separate variables, but the cartesian product of those domains. That is, it is a set of tuples, each of which represents a configuration (which may or may not be valid.)

$$\begin{aligned} \llbracket \text{define } x:v_1, \dots, v_n \rrbracket &= (\langle x \rangle, \{v_1, \dots, v_n\}) \\ \llbracket \text{defn}_1; \dots; \text{defn}_n \rrbracket &= (X_1 \cdot \dots \cdot X_n, D_1 \times \dots \times D_n) \\ \text{where} \\ (X_i, D_i) &= \llbracket \text{defn}_i \rrbracket \end{aligned}$$

$$\llbracket \mathbf{export} \ x_1, \dots, x_n \rrbracket = \langle x_1, \dots, x_n \rangle$$

$$\llbracket \mathit{expt}_1; \dots; \mathit{expt}_n \rrbracket = (X_1^E \cdot \dots \cdot X_n^E)$$

where

$$X_i^E = \llbracket \mathit{expt}_i \rrbracket$$

The next function obtains the sequence of variables and the corresponding domain that are added to a module because of the exported variables in the imported module. The function \mathcal{N} is used to look up the information on the imported module (N), which is a tuple, and we use the notation X_N and D_N to select the X and D fields from that tuple.

$$\llbracket \mathbf{import} \ \mathit{modname} \ \mathbf{as} \ \mathit{instname} \ \mathbf{if} \ \mathit{cond} \rrbracket_{\text{XD}} \mathcal{N} = (X, D)$$

where

$$N = \mathcal{N}(\mathit{modname})$$

$$X = \mathit{addprefix}(\mathit{instname}, X_N^E)$$

$$D = D_N^E$$

$$\llbracket \mathit{impt}_1; \dots; \mathit{impt}_n \rrbracket_{\text{XD}} \mathcal{N} = (X_1 \cdot \dots \cdot X_n, D_1 \times \dots \times D_n)$$

where

$$(X_i, D_i) = \llbracket \mathit{impt}_i \rrbracket_{\text{XD}} \mathcal{N}$$

Then, we define the meaning of a module (as far as names are concerned) by the following function, which takes the contributions to the variables and domain from the different sources (local definitions, and imported definitions), and puts that information into a 4-tuple. Note that this semantic function for a module can be seen as returning a mapping from the function \mathcal{N} (which maps module names to these 4-tuples) to a pair mapping the present module name to such a 4-tuple.

To examine the function more closely, the first two subequations define the local and imported contributions to the sequence of variable names and to the domain. The next two subequations define the composition of this information¹. The following two define what part of this information pertains to variables that are exported. This will be taken into account in interpreting the importing of these modules in the next fixpoint iteration. The last line, finally, constructs the 4-tuple N .

¹ The operators here are to be understood as simply composing their operands flatly, that is, not to introduce hierarchical structure.

$\llbracket \mathbf{module} \text{ } modname; expts; impts; defns; ensns \rrbracket_{XD} \mathcal{N} = (modname, N)$

where

$$\begin{aligned} (X_L, D_L) &= \llbracket defns \rrbracket \\ (X_I, D_I) &= \llbracket impts \rrbracket_{XD} \mathcal{N} \\ X &= X_L \cdot X_I \\ D &= D_L \times D_I \\ X^E &= \llbracket expts \rrbracket \\ D^E &= projectset(X, X^E, D) \\ N &= (X, D, X^E, D^E) \end{aligned}$$

$\llbracket modl_1; \dots; modl_n; \rrbracket_{XD} \mathcal{N} = \{P_1, \dots, P_n\}$

where

$$P_i = \llbracket modl_i \rrbracket_{XD} \mathcal{N}$$

Finally, the meaning of a modular configuration problem posed as a sequence of modules, is given as a function from the module names to the 4-tuples containing the information pertaining to variable sequences, the domains of these modules, and the parts of those that are exported. This function is given by:

$$\mathcal{N} = \mu F_N. \llbracket \mathcal{M} \rrbracket_{XD} F_N$$

2.5 Compilation of Solutions

The second phase in the compilation of a modular configuration problem consists of the computation of the solution sets of all modules, and the conditions for the instantiations of those modules. In order to maintain global consistency, and since modules can import each other recursively, this needs to be computed by a fixpoint iteration.

We begin with the functions that obtain the subset of configurations that satisfy constraints (S), given the sequence of variable names and the domain:

$\llbracket \mathbf{ensure} \ e \rrbracket XD = \llbracket e \rrbracket$

where

$$\begin{aligned} \langle x_1, \dots, x_n \rangle &= X \\ \llbracket x = v \rrbracket &= \{(v_1, \dots, v_n) \in D \mid x_i = x, v_i = v\} \\ \llbracket e_1 \wedge e_2 \rrbracket &= \llbracket e_1 \rrbracket \cap \llbracket e_2 \rrbracket \\ \llbracket e_1 \vee e_2 \rrbracket &= \llbracket e_1 \rrbracket \cup \llbracket e_2 \rrbracket \\ \llbracket \neg e \rrbracket &= D \setminus \llbracket e \rrbracket \end{aligned}$$

$\llbracket ensn_1; \dots; ensn_n \rrbracket XD = (S_1 \cap \dots \cap S_n)$

where

$$S_i = \llbracket ensn_i \rrbracket XD$$

Next, we have the functions that obtain the constraints coming from submodules (S) and the instantiation conditions for those submodules (Δ). Note the equation that defines S , which reflects the fact that constraints from submodules are taken into account if and only if that submodule exists. (That is, when the final configuration is known to be an element of C .) Also, note the function \mathcal{A} which we use to look up the information about the imported module.

$$\llbracket \mathbf{import} \text{ } modname \text{ as } instname \text{ if } cond \rrbracket_{S\Delta} XDA = (S, \Delta)$$

where

$$\begin{aligned} \langle x_1, \dots, x_n \rangle &= X \\ \llbracket x = v \rrbracket &= \{(v_1, \dots, v_n) \in D \mid x_i = x, v_i = v\} \\ \llbracket c_1 \wedge c_2 \rrbracket &= \llbracket c_1 \rrbracket \cap \llbracket c_2 \rrbracket \\ \llbracket c_1 \vee c_2 \rrbracket &= \llbracket c_1 \rrbracket \cup \llbracket c_2 \rrbracket \\ \llbracket \neg c \rrbracket &= D \setminus \llbracket c \rrbracket \\ C &= \llbracket cond \rrbracket \\ A &= \mathcal{A}(modname) \\ X_S &= addprefix(instname, X_A^E) \\ S &= (D \setminus C) \cup embedset(X, X_S, D, S_A^E) \\ \Delta &= \{(C, instname, modname)\} \end{aligned}$$

$$\llbracket impt_1; \dots; impt_n \rrbracket_{S\Delta} XDA = (S_1 \cap \dots \cap S_n, \Delta_1 \cup \dots \cup \Delta_n)$$

where

$$(S_i, \Delta_i) = \llbracket impt_i \rrbracket_{S\Delta} XDA$$

Then, we define the meaning of a module by the following function. It simply looks up the X , D , X^E and D^E components in \mathcal{N} , the Δ component follows from the module imports only, and the S component is taken from the constraints and the imported modules. (Constraints from parent modules are handled at runtime.)

$$\llbracket \mathbf{module} \text{ } modname; expts; impts; defns; ensns \rrbracket_{S\Delta} \mathcal{A} = (modname, A)$$

where

$$\begin{aligned} (X, D, X^E, D^E) &= \mathcal{N}(modname) \\ S_L &= \llbracket ensns \rrbracket XD \\ (S_I, \Delta) &= \llbracket impts \rrbracket_{S\Delta} XDA \\ S &= S_L \cap S_I \\ S^E &= projectset(X, X^E, S) \\ A &= (X, D, S, \Delta, X^E, D^E, S^E) \end{aligned}$$

$$\llbracket modl_1; \dots; modl_n \rrbracket = \{P_1, \dots, P_n\}$$

where

$$P_i = \llbracket modl_i \rrbracket_{S\Delta} \mathcal{A}$$

Finally, the meaning of a modular configuration problem posed as a sequence of modules, is given as a function from the module names to 7-tuples containing

information about the variable sequences, domains, solution sets, instantiation conditions, and those parts of them that are relevant for the exported variables. This function is given by:

$$\mathcal{A} = \mu F_A. \llbracket \mathcal{M} \rrbracket_{S\Delta} F_A$$

2.6 Execution

User interaction with the configurator takes place during what we call the execution phase. At this time, the configuration problem has been compiled into the function \mathcal{A} , mapping module names to 7-tuples.

During user interaction, we will represent the configurator state by a 4-tuple (I, A, R, T) , in which T is itself a set of such 4-tuples. This is the way in which we represent the tree of module instances. This tree is always of finite size (the proof is beyond the scope of this paper). The other elements of the tuple are: I , which is the name of the instance; A , which is the 7-tuple that corresponds with the module the instance was created from; and R , which is the set of remaining valid solutions for this instance.

First, we will look at two auxiliary functions which help with instantiation and synchronization, and after that, we will look at the initial state, choices, and update functions that we need to operate on the configuration states.

So we begin with the first auxiliary function, which takes care of instantiation of subinstances. It takes the definition of a module (A), the remaining solution set (R) and the current set of subtrees (T), and returns the new set of subtrees, keeping all current subtrees, and adding all subinstances which are in the set Δ_A , for which the instantiation condition is satisfied, but which do not yet exist in the current set of subinstances. Note that the new instances are created with a full set of solutions, and without any subinstances of their own. This task is left to all functions that use this function. Also note that it does not take a full (I, A, R, T) tuple as an argument - this is because the semantics turn out to be slightly less tedious this way.

$$subtrees(A, R, T) = T \cup \{(I, \mathcal{A}(M), S_{\mathcal{A}(M)}, \emptyset) \mid (C, I, M) \in \Delta_A, R \subseteq C, \forall N \in T. I_N \neq I\}$$

Then we continue with the second auxiliary function, which takes care of the downward synchronization of instances in the instance tree, and of the resulting instantiations by using *subtrees*. That is to say, when the solution set of an instance is made smaller, this function can be used to ensure the consistency of that instance with all its subinstances, and their subinstances, and so on.

$$\begin{aligned}
\text{propagate}(X_P, R_P, (I, A, R, T)) &= (I, A, R', T') \\
\text{where} \\
X_S &= \text{addprefix}(I, X_A^E) \\
R_S &= \text{projectset}(X_P, X_S, R_P) \\
R' &= R \cap \text{embedset}(X_A, X_A^E, D_A, R_S) \\
T' &= \{\text{propagate}(X_A, R', N) \mid N \in \text{subtrees}(A, R', T)\}
\end{aligned}$$

We now turn to the main thrust of the semantics of execution of a modular configuration problem.

The first function we turn our attention to is the *initialstate* function. The *initialstate* function takes a module name, and returns a tuple representing a configuration state. This tuple is a 4-tuple representing the root instance.

$$\begin{aligned}
\text{initialstate}(\text{modname}) &= (I, A, R, T) \\
\text{where} \\
I &= \epsilon \\
A &= \mathcal{A}(\text{modname}) \\
R &= S_A \\
T &= \{\text{propagate}(X_A, R, N) \mid N \in \text{subtrees}(A, R, \emptyset)\}
\end{aligned}$$

Then, we turn to the operation of finding out what the choices are given a particular configuration state. We have two functions for this purpose: *localchoices* and *globalchoices*. One could consider *localchoices* to be an auxiliary function, as it only generates the choices for the root-instance of the tree. It is used by *globalchoices*, which recursively builds up the set of possible choices for the entire instance tree.

$$\begin{aligned}
\text{localchoices}(I, A, R, T) &= \{(x_1, D_1), \dots, (x_n, D_n)\} \\
\text{where} \\
\langle x_1, \dots, x_n \rangle &= X_A \\
D_i &= \{v_i \mid (v_1, \dots, v_n) \in R\}
\end{aligned}$$

$$\begin{aligned}
\text{globalchoices}(I, A, R, T) &= \text{localchoices}(I, A, R, T) \cup \text{subtreechoices} \\
\text{where} \\
\text{subtreechoices} &= \{(I_N \cdot \langle \cdot \rangle \cdot x, D) \mid N \in T, (x, D) \in \text{globalchoices}(N)\}
\end{aligned}$$

And to finish, we consider the *update* functions. There are two versions. The first deals with the case where a simple variable name is given, in which case the current instance needs to be updated, and the change needs to be propagated to the subinstances. The second deals with the case where the variable is a qualified variable. In this case the subinstance it belongs to needs to be updated first, then the current instance needs to resynchronize with that subinstance, and finally all other subinstances need to resynchronize with the current instance.

The idea is that the correct version of the function to be used is to be determined by pattern matching. If both versions match, which happens with shared variables, either can be used, since shared variables are always kept synchronized.

The first version of the *update* function is for the case where the variable name x is a simple name. In this case, it is the current instance that needs to be modified, after which its subtrees are to be synchronized again. The former is done by cutting down R , and then the latter is done by using a combination of the *subtrees* and *propagate* functions.

$$\text{update}(\langle x \rangle, v, (I, A, R, T)) = (I, A, R', T')$$

where

$$\begin{aligned} \langle x_1, \dots, x_n \rangle &= X_A \\ R' &= R \cap \{(v_1, \dots, v_n) \in D_A \mid x_i = x, v_i = v\} \\ T' &= \{\text{propagate}(X_A, R', N) \mid N \in \text{subtrees}(A, R', T)\} \end{aligned}$$

The second version of the *update* function is for the case where the variable name x is prefixed with a path. In this case, we take off the front of the name, which is the instance name of the submodule to which this selection is to be routed. We find the subtree N which has this instance name, and update it recursively. When this is done, we need to synchronize the current module with the new subtree N' , and after that, we need to synchronize the other subtrees with the state of the current module.

$$\text{update}(I \cdot \langle \cdot \rangle \cdot x, v, (I, A, R, T)) = (I, A, R', T')$$

where

$$\begin{aligned} \{N\} &= \{N \in T \mid N_I = I\} \\ N' &= \text{update}(x, v, N) \\ X_S &= \text{addprefix}(I_N, X_{A_N}^E) \\ R_S &= \text{projectset}(X_{A_N}, X_{A_N}^E, R_{N'}) \\ R' &= R \cap \text{embedset}(X_A, X_S, D_A, R_S) \\ T' &= N' \cup \{\text{propagate}(X_A, R', N'') \mid N'' \in \text{subtrees}(A, R', T) \setminus \{N\}\} \end{aligned}$$

3 Conclusion

In this paper, we have extended the two-stage approach compiling into BDDs with the notion of modules. We allow for the combining of such modules by means of shared variables, for conditional instantiation of such modules, and for recursion between such modules. For the resulting configuration system, we guarantee the user that she can select any configuration that is valid, none that are not valid, and that she can always find a finite configuration. In the near future, we will complete the modeling of a real-world modular configuration problem. The results of that work will guide us in our further research.

References

1. Sabin, D., Weigel, R.: Product configuration frameworks - a survey. *Intelligent Systems, IEEE* **13** (1998) 42–49
2. Tiihonen, J., Soinen, T., Niemelä, I., Sulonen, R.: Empirical testing of a weight constraint rule based configurator. In: *ECAI 2002 Configuration Workshop*. (2002) 17–22 <http://www.soberit.hut.fi/pdmg/Empirical/index.html>.
3. Amilhastre, J., Fargier, H., Marquis, P.: Consistency restoration and explanations in dynamic CSPs-application to configuration. *Artificial Intelligence* **1-2** (2002) 199–234 <ftp://fpt.irit.fr/pub/IRIT/RPDM/Configuration/>.
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