Efficient Interactive Configuration of Unbounded Modular Systems

Erik Roland van der Meer
IT University of Copenhagen
ervandermeer@itu.dk

Andrzej Wasowski
IT University of Copenhagen
wasowski@itu.dk

Henrik Reif Andersen
IT University of Copenhagen
hra@itu.dk

ABSTRACT

Interactive configuration guides a user searching through a large combinatorial space of solutions to a system of constraints. We investigate a class of very expressive underlying constraint satisfaction problems: modular recursive constraint systems of unbounded size. A precomputation step is used to obtain a configuration algorithm for such systems that supports the user efficiently with bounded response time. This precomputation step determines all solutions for each module, which are computed and stored in compact data structures such as Binary Decision Diagrams (BDDs), in order to eliminate run-time search. The precomputation step also detects ill-behaved module collections that have no finite solutions. The runtime interaction algorithm scales well as its response time only depends on the amount of the information passed locally between the modules, and not on the size of the entire configured structure. Our algorithm was implemented and tested on an industrial example, and gives good response times. We believe this is the first known sound and complete algorithm for solving unbounded interactive configuration problems, with bounded response time per interaction.

Categories and Subject Descriptors

F.4.1 [**Mathematical Logic**]: Logic and constraint programming; I.2.1 [**Artificial Intelligence**]: Applications and Expert Systems

General Terms

Algorithms, Languages, Performance

Keywords

Interactive configuration, Constraint satisfaction

1. INTRODUCTION

Difficulty in configuring complex software or devices is a major obstacle in increasing sales of many technologically advanced products. Computer assisted configuration gains attention, as many products become more customizable and more complex. Interactive configurators are applied in electronic sales, power supply restoration, customization of controllers for embedded systems, ERP software, etc.

An interactive configurator assists its user in her search through a combinatorial space of configurations for a given wanted solution. The search progresses by repeated choices of values for variables in the underlying Constraint Satisfaction Problem. The guidance takes the form of computing projections of the future valid choices down to the domains of the variables in the CSP, based on the already made choices. In this way the supporting program offers its user only valid choices for all variables throughout the configuration process. The underlying problem is that of deciding satisfiability and unsatisfiability for propositional logic and is therefore NP-hard and co-NP-hard. For systems of fixed size, a precomputation step can factor the NP-hardness out of the problem so that only a polynomial problem is solved during user interaction, even if the solution space is exponentially large [5, 10, 4]. But restriction to fixed size and structure precludes modeling of systems built of an arbitrary number of components, such as power-supply equipment for data centers, air conditioning or cooling installations, storage furniture, pumping systems, etc. In all these examples any concrete installation is finite and has a fixed structure; but in general there is no limit on the size of structures that can be constructed. Such systems cannot at all be cast as classical CSP problems involving a finite number of variables over finite domains. Traditional CSP models can only represent a finite number of legal configurations whereas the examples mentioned previously have virtually infinitely many solutions.

Recursion is the way typically used in programming language theory to express unboundedness. We show how the precomputation method can be extended to a rich class of unbounded CSP-problems able to represent the applications mentioned above. The problems investigated are described by recursive modules that can generate CSP-instances of unbounded size. Our interactive modular configuration algorithms enjoy guaranteed response times, which can be precomputed offline. Crucially for scalability, the response time only depends on the amount of information passed locally between neighbor modules, and is independent of the size of the entire tree. According to our knowledge this is the first complete algorithm for solving an unbounded interactive configuration problem, with bounded response time. The precomputation algorithm also identifies ill-behaved solution subspaces that would necessarily lead to unbounded instances. Our algorithms have been implemented, and show very good performance on a case study from the air conditioning industry.

Consider an example of a modular recursive configuration problem representing a simplified USB tree consisting of hubs, printers, and exactly one keyboard. It is given in the source language of our prototype modular configurator. The main module `usb` describes a usb port, which can be unused, used to connect a keyboard, or a printer, or may be a hub, which opens two new usb ports. The type of the module is modeled by the `type` variable with the values `unused`, `keyboard`, `printer`, and `hub` in its domain. The variable `keybd` models the presence of a keyboard on the bus. It indicates whether or not the tree rooted at the given instance of the module contains a keyboard.

```
module usb;

define type : unused, keyboard, printer, hub;
define keybd : yes, no;
export keybd;

import keyboard if type = keyboard;
import printer  if type = printer;
import usb as hub1 if type = hub;
import usb as hub2 if type = hub;

ensure type = unused   -> keybd = no ;
ensure type = keyboard -> keybd = yes;
ensure type = printer  -> keybd = no ;
ensure type = hub -> ((hub1.keybd=no |hub2.keybd=no ) &
     (keybd = yes <-> (hub1.keybd=yes|hub2.keybd=yes)));
```

The `usb` module imports the `keyboard`, `printer`, and `usb` modules depending on the value of `type`. Two instances of module `usb`, called `hub1` and `hub2`, are imported recursively when `type = hub`. The constraints (`ensure`) define the behavior of `keybd`. In particular, the last one states that at most one port has a keyboard, and that a hub has a keyboard if one port does. These are standard CSP constraints.

For simplicity of the example, the keyboard and printer modules are kept empty, basically stating that we are interested only in presence of the respective devices, and not in their specific properties. In a more realistic model one could require that the keyboards and printers were of a particular kind, for instance supporting the communication over USB, thus introducing structure also inside these basic modules.

```
module keyboard;
module printer;
```

Finally, a root module is added, ensuring that there is exactly one keyboard in the entire configuration tree:

```
module root;
import usb;
ensure usb.keybd = yes;
```

The configurator initiates the interaction by creating a root node, with the only variable `usb.keybd` set to `yes` (automatically inferred from the constraint) and the unconditionally imported `usb` submodule instantiated. The user can choose to configure the currently focused module `root`, but since all its variables are assigned, a better choice is to *move* the focus to the `usb` child, where values for variables can still be chosen. If just a keyboard is required, *assign* `type = keyboard` and the keyboard submodule is automatically instantiated, finalizing the configuration. Assignment of $type = printer$ is not allowed by the tool, as the configuration created in this way does not fulfill the constraint that there is exactly one keyboard in the system.

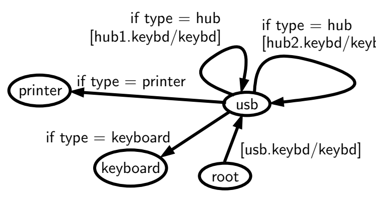

Figure 1: A multigraph for the USB example

The most essential properties of our configurator are: (1) only valid choices are allowed (no risk of global inconsistencies), (2) as soon as the instantiation constraint of a module is satisfied, the submodule is instantiated, and (3) all possible finite configurations can be constructed. In effect, the user does not risk being restricted in his choices, while being free from the annoyances of backtracking.

We formalize the problem in section 2. The factoring algorithm is presented in section 3. The interactive algorithms are shown and analyzed in sections 4–5. Section 6 discusses related work and concludes.

2. MODULAR CONFIGURATION

We shall now precisely define the modular configuration problem and the nature of its solutions.

Definition 1. *A standalone constraint satisfaction problem (CSP) is a triple $P(X, D, C)$ comprising a finite totally ordered set of variables $X = \{x_1, \ldots, x_m\}$, a finite set of corresponding finite domains $D = \{d_1, \ldots, d_m\}$, and a finite set of constraints $C = \{c_{Y_1}, \ldots, c_{Y_m}\}$. A constraint c_Y has a scope $Y = \{y_1, \ldots, y_k\} \subseteq X$, and is a finite set of tuples $c_Y \subseteq d_{y_1} \times \cdots \times d_{y_k}$.*

We denote the projection of a tuple s onto a set of variables Y by $s{\downarrow}Y$. We also lift the $\downarrow$ operator to sets of tuples.

Definition 2. *An assignment for a CSP $P(X, D, C)$ is a tuple from the product domain $d_1 \times \cdots \times d_m$. An assignment s satisfies a constraint $c_Y \in C$ iff $s{\downarrow}Y \in c_Y$. A solution to P is an assignment satisfying all the constraints in C.*

A *modular* configuration problem is a directed multigraph, where vertices represent standalone CSPs (modules) and edges represent imports. The edges are directed from importing toward imported modules. The graph may contain cycles, loops, and multiple edges between any two vertices. Edges are labeled by instantiation constraints and interfaces (shared variables of incident modules). We write $\mathrm{src}\, e$ and $\mathrm{dst}\, e$ for the source and the destination vertex of e respectively. A graph is rooted if it has a designated root vertex.

Definition 3. *A modular configuration problem is a tuple $M(G(V, E, v_0), P, I, J)$, where $G(V, E, v_0)$ is a finite directed multigraph rooted in v_0. P is a map associating each vertex $v \in V$ with its standalone CSP $P_v(X_v, D_v, C_v)$, I associates each edge $e \in E$ with its instantiation constraint I_e, while J associates e with its interface J_e—a partial bijective map $J_e = [x_1 \mapsto y_1, \ldots, x_k \mapsto y_k]$ between shared variables of $\mathrm{src}\, e$ and $\mathrm{dst}\, e$: $\{x_i\}_{i=1..k} \subseteq X_{\mathrm{src}\, e}$, and $\{y_i\}_{i=1..k} \subseteq X_{\mathrm{dst}\, e}$.*

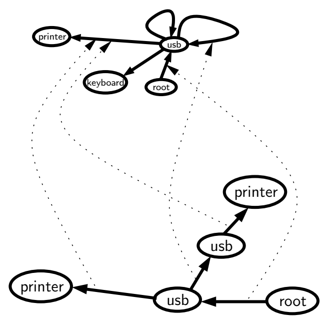

Figure 2: A realization of the USB example graph

Fig. 1 shows the multigraph for the usb example. The instantiation constraint for one of the self loop edges on the `usb` vertex is the set of tuples satisfying `type = hub`, and a `usb` module is instantiated when this constraint is satisfied.

An interface J_e relates the names of the variables in two neighbouring standalone CSPs. Once e has been instantiated, the shared variable assignment of its source must be the same as the target variable assignment modulo the renaming of J_e. In the usb example, qualified names are used to define J_e implicitly. A variable `usb.keybd` in the `root` module is called `keybd` in its `usb` submodule. Note that there is a real need for renaming variables across modules, to distinguish between variables of several imported children of the same type (like `hub1.keybd` and `hub2.keybd`).

Though technically an interface is a map between variable sets, we often use it as two projections and a permutation, which allows direct applications to sets of assignments. If $e = (v_1, v_2)$, we write $A{\downarrow}J_{v_1}$ for a projection of sets of tuples A on variables in the domain of J_e and $A{\downarrow}J_{v_2}$ for projection of A on variables in the range of J_e. Similarly, we apply J_e as a permutation indicating the direction of renaming. For example $J_{v_1 \to v_2}(A{\downarrow}J_{v_1})$ means that the set of tuples A is first restricted to the variables of v_1 that are shared with v_2, and then each tuple is permuted, becoming a projection of an assignment to a shared subset of X_{v_2}. We take the freedom to indicate the direction of permutation and projection sides with whatever unambiguously indicates it in the context: vertices in a graph, nodes in a configuration tree, or names of sets of tuples.

A *configuration* is a tree with nodes and arcs created by unfolding the problem multigraph:

Definition 4. *Let $T(N, A)$ be a tree, $G(V, E, v_0)$ be a directed rooted multigraph, t_N be a map from the set of nodes N to the set of vertices V, and t_A be a map from the set of arcs A to the set of edges E. We say that the triple (T, t_N, t_A) is a realization of G, written $T \sqsubseteq_{t_N,t_A} G$, iff:*

1. (t_N, t_A) is a homomorhpism from T to G: for each arc $a \in A$ it holds that $t_N(\operatorname{src} a) = \operatorname{src} t_A(a)$, $t_N(\operatorname{dst} a) = \operatorname{dst}\, t_A(a)$

2. t_A restricted to the arcs adjacent to any single node n ($t_A{\downarrow}_{\operatorname{adj} n}$) is an injection (so t_A is locally injective).

The first requirement ensures that T is a proper unfolding of G (can be generated from G), while the second makes sure that for each of the nodes in T there is only one neighbour of a particular kind (at most one parent and at most one child corresponding to each child in G).

Consider the example of Fig. 2. The tree in the lower part of the figure is a correct realization of the previously

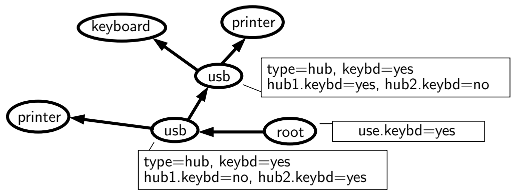

Figure 3: A configuration of the USB model

discussed problem multigraph for the USB example. The t_N map is implicitly represented by the names of the nodes, i.e. each node in the tree is mapped to a node in the graph with the same name. The arc map t_A is visualized by means of dotted arrows.

Definition 5. *A configuration of a modular configuration problem $M(G(V, E, v_0), P, I, J)$ is a tuple $\phi(T(N, A), t_N, t_A, s)$, where s maps each node n to a solution s_n of the standalone CSP $P_{t_N(n)}$ and:*

[RL] $T \sqsubseteq_{t_N,t_A} G$ and $\exists n_0 \in N$ such that $t_N(n_0) = v_0$.

[LC] All nodes are locally consistent: $\forall n \in N.\ s_n \in \operatorname{Sol} P_{t_N(n)}$

[AC] ϕ is arc consistent:

$$\forall a(n_1, n_2) \in A.\ J_{n_1 \to n_2}(s_{n_1}{\downarrow}J_{n_1}) = s_{n_2}{\downarrow}J_{n_2}$$

[WF] ϕ is well-formed: $\forall n_1 \in N.\ \forall e \in \operatorname{out} t_N(n_1).\ s_{n_1} \in I_{t_A(a)}$ iff $\exists n_2 \in N. t_N(n_2) = \operatorname{dst} e$ and there is an arc $a(n_1, n_2) \in A$

A configuration is finite iff the set of nodes N is finite.

Figure 3 presents one possible configuration of the USB model. It models a USB hub with one printer connected to the first port and another hub connected to the second port. A keyboard and a printer are connected to the latter. Observe that there are more nodes in the configuration tree than in the original problem graph. Values of variables, for modules that have variables, are presented in boxes adjacent to respective nodes. The arc map, t_A, is not shown in order to avoid clutter.

Notice that the realization tree of Fig. 2 is not a proper configuration. First, it lacks information about the variable assignments—in fact no consistent variable assignment for that tree exists due to the lack of the keyboard node. Second, it was not properly instantiated—the second `usb` node on the path from the tree was missing one child instantiaton. The model requires two sub modules for any hub, while this node only has one.

3. OFFLINE PRECOMPUTATION

Let us now introduce two auxiliary notions involving interfaces and a central property of directional consistency:

Definition 6. *If A, B are sets of assignments, $s \in A$, and J is an interface between them, then $s \in_{J_{A \to B}} B$ iff $\exists s' \in B.\ J_{A \to B}(s \downarrow_{J_A}) = s' \downarrow_{J_B}$.*

Definition 7 (Semi-Join). *If A, B are sets of assignments, $s \in A$, and J is an interface between them, then $A \ltimes_{J_{A \to B}} B := \{s \in A \mid s \in_{J_{A \to B}} B\}$.*

Definition 8 (Directional Arc Consistency). *Let A and B be sets of assignments, $s \in A$, and J an interface between them. Then $A \subseteq_{J_{A \to B}} B$ iff $\forall s \in A.\ s \in_{J_{A \to B}} B$.*

Intuitively, $A \ltimes_{J_{A\to B}} B$ is the largest subset of A, such that each of its members has a compatible assignment in B, while $A \subseteq_{J_{A\to B}} B$ means that each assignment in A has a compatible assignment in B.

Compilation finds, for each standalone CSP, the subset of solutions to that CSP that can be used in valid configurations with that CSP at the root. The COMPILE algorithm first solves the separate CSPs in the configuration problem, and stores them in the SP_v sets. Then it determines which of these solutions can be used in finite configurations that satisfy all contraint types. The algorithm relies on a minimum fixpoint computation, which incrementally adds assignments for which these properties can be established:

```
COMPILE($M(G(V,E,v_0),P,I,J)$)
1  for each $v \in V$ do $SP_v \leftarrow \mathrm{Sol}(P_v)$
2                        $S_v \leftarrow \emptyset$
3  repeat  for each $v \in V$ do $S'_v \leftarrow S_v$
4          for each $v \in V$ do $S_v \leftarrow \mathrm{step}(v)$
5  until $S = S'$
6  return $S$
```

where the function step(v) is defined as:

$$\mathrm{step}(v) = \bigcap_{e(v,v')\in \mathrm{out}\, v} ((SP_v \cap I_e) \ltimes_{J_{v\to v'}} S'_{v'}) \cup (SP_v \setminus I_e) \qquad (1)$$

The solutions to the CSP of a module v can enter the solution set of that module (S_v) iff there already are compatible solutions in the solution sets of all the submodules that they would instantiate. The step function is monotonic, and is iterated over a finite lattice, which guarantees termination.

The compiler warns the designer if any of the sets of solutions S_v computed by COMPILE is empty. If $S_v = \emptyset$ then module v can never be instantiated at runtime, which hints at a bug in the product model. Moreover, if the root set S_{v_0} is empty, then no interaction is possible and no configuration can be created at all: the compiler fails with an error.

4. INTERACTIVE CONFIGURATION

The configurator initializes the global data structures using INIT. The user can find out what options are still open for each variable in the current *focus* node using VALID-DOMAINS. She can then assign permitted values using ASSIGN, and move the focus using MOVE. The algorithms delegate instantiation to INSTANTIATE. They assume that the configuration problem $M(G(V,E,V_0),P,I,J)$ and the compilation result S are globally visible. They also assume that the following mutable data structures are globally available: a tree $T(N,A)$, the maps t_N, t_A, and R, and the focus f.

```
INIT($M(G(V,E,v_0),P,I,J)$)
1  Empty tree $T(N,A)$, empty maps $t_N$, $t_A$, $R$
2  Create a node $n_0$ in $T$
3  $t_N(n_0) \leftarrow v_0$
4  $R_{n_0} \leftarrow S_{v_0}$
5  $f \leftarrow n_0$
6  INSTANTIATE()
```

```
VALID-DOMAINS()
1  Let $X = \{x_1 \ldots, x_k\}$ be the variables of $f$ (in order)
2  return $(R_n{\downarrow}x_1, \ldots, R_n{\downarrow}x_k)$
```

```
ASSIGN ($x$: variable in $f$, $v$: value in the valid domain of $x$)
1  $R_f \leftarrow \{s \in R_f \mid s_x = v\}$
2  INSTANTIATE()
```

```
MOVE ($n$: node adjacent to $f$)
1  $R_n \leftarrow R_n \ltimes_{n\to f} R_f$
2  $f \leftarrow n$
3  INSTANTIATE()
```

```
INSTANTIATE()
1  for each edge $e \in E$ from $t_N(f)$
2      do if $R_f \subseteq I_e$ and $\forall a \in A$ from $f$, $t_A(a) \neq e$
3            then create a node $n'$ in $T$
4                 create an arc $a(f,n')$ in $T$
5                 $t_N(n') \leftarrow \mathrm{dst}\, e$
6                 $t_A(a) \leftarrow e$
7                 $R_{n'} \leftarrow S_{\mathrm{dst}\, e}$
```

The algorithms do not rely on any particular set representation (list of tuples, cartesian product representation, decision diagrams, etc). Our implementation uses binary decision diagrams (BDDs) [3], which in our experience are good for representing sets arising in configuration. Below we discuss the complexity of our BDD-based implementation.

BDDs only represent constraints over binary variables directly. Variables with bigger finite domains are modeled by sets of binary variables. Let b_v denote the number of bits needed to represent variables involved in all interfaces incident with vertex v, counting variables shared among interfaces only once. In practice b_v tends to small for well structured models, as they usually pass very little information across the nodes. The maximum size of a BDD representing a set of interface tuples is: $\mathbb{J}_v := O(2^{b_v})$.

Let $\mathbb{S}_v$ denote the size of the BDD representing the set S_v. The sizes of BDDs representing the R_n sets during interaction are bound by $\mathbb{R}_n := O(2^{b_{T_N(n)}} \cdot \mathbb{S}_{t_N(n)})$, as any of these BDDs can be created by a conjunction of a BDD containing b_v variables and a BDD of size $\mathbb{S}_v$. This constant can be big, but after compilation it is known.

We will denote the maximum size of a BDD representing the instantiation constraint for an import in node n as $\mathbb{I}_n$. The dominating operation of INSTANTIATE is the inclusion test in line 2. It is implemented by constructing the BDD for $R_f \implies I_e$ and checking it for tautology. The tautology check is constant time, but the construction of the implication is linear in the size of both R_f and I_e. As this is repeated for every potential submodule of the focus node f, the running time of INSTANTIATE is $O(\mathbb{R}_f \mathbb{I}_f \cdot |\,\mathrm{out}\, t_N(f)|)$, where all the components are statically known. Since both $\mathbb{I}_f$ and the number of imports in a single module tend to be small, the running time is dominated by $\mathbb{R}_f$.

The first line of MOVE can be implemented by the simultaneous existential quantification, substitution, and conjunction. Existential quantification is linear in the size of the original BDD ($\mathbb{R}_f$) and exponential in the number of BDD variables remaining after the quantification (bound by b_n). Denote the bound on this operation by: $\mathbb{E}_{f\to n} := O(2^{b_{f\to n}})$, where $b_{f\to n}$ is the number of bits on the interface between f and n. Again, $\mathbb{E}_{f\to n}$ is small if the interfaces are narrow.

Substitution can be implemented in linear time for the specific use in the above algorithms. The cost of conjunction is linear in the size of both operands: $O(\mathbb{J}_f \mathbb{R}_f)$. Finally, the worst case running time of MOVE(n), including its call to

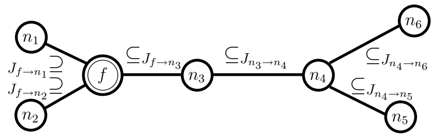
n_1
n_2
f
n_3
n_4
n_5
n_6
$J_{f \to n_1} \supseteq$
$J_{f \to n_2} \supseteq$
$\subseteq J_{f \to n_3}$
$\subseteq J_{n_3 \to n_4}$
$\subseteq J_{n_4 \to n_6}$
$\subseteq J_{n_4 \to n_5}$

Figure 4: DAC holds from the focus f outwards.

INSTANTIATE, is $O(\mathbb{R}_f \mathbb{E}_{f \to n} + \mathbb{J}_f \mathbb{R}_f + \mathbb{R}_f \mathbb{I}_n \cdot |\operatorname{out} t_N(f)|)$. Observe that MOVE operates entirely locally. Its running time is independent of the size of the entire configuration problem, the global topology of the network, etc. So if the configuration problem is well structured, we can give good time bounds for the responsiveness of the user interface.

The restriction operation implementing the first line of the call to ASSIGN(x, v) is realized by the BDD restrict operation, which is linear in the size of the BDD representing the solution space (bounded by $\mathbb{R}_{t_N(n)}$). Consequently, the worst-case running time of ASSIGN is dominated by the running time of INSTANTIATE, and is equal to $O(\mathbb{R}_{t_N(f)} + \mathbb{R}_{t_N(f)} \mathbb{I}_{t_N(f)} |\operatorname{out} t_N(f)|) = O(\mathbb{R}_{t_N(f)} \mathbb{I}_{t_N(f)} |\operatorname{out} t_N(f)|)$.

The user interface wants to query the engine for valid domains of variables each time an assignment or a move is made. A naive $O(\mathbb{R}_{t_N(f)} \cdot \sum_{v \in X_{t_N(f)}} \mid d_v \mid)$ BDD algorithm for VALID-DOMAINS can be implemented by performing a tentative assignment (without instantiating) and testing satisfiability for each value in the focus node. Improvements to this algorithm are known and we use one of them.

Observe again that the bounds are exponential in constants, which makes it possible to establish the feasibility of using the tool for a given problem before testing it on actual users. Guaranteed response times can be computed. The response does not become slower with the growth of the tree being constructed. All complexity bounds depend on local parameters only, and not on the global structure.

5. CORRECTNESS

We assume that the problem M has been annotated with solution sets S_v returned from the compilation process. Partial configurations of annotated problems are built by consecutive applications of MOVE and ASSIGN algorithms.

DEFINITION 9. *A partial configuration of an annotated modular configuration problem $M(G(V, E, v_0), P, I, J, S)$ is a tuple $\gamma(T(N, A), t_N, t_A, R, f)$, where $f \in N$ is the current focus node, R maps nodes to sets of remaining solutions, and:*

[RL] Realization: $T \sqsubseteq_{t_N, t_A} G$

[LC] Local consistency: $\forall n \in N. R_n \subseteq S_{t_N(n)}$ and $R_n \neq \emptyset$

[DAC] Directional arc consistency: $\forall a(n_1, n_2) \in A$. if n_1 is closer to f than n_2 then $R_{n_1} \subseteq_{J_{n_1 \to n_2}} R_{n_2}$ else $R_{n_2} \subseteq_{J_{n_2 \to n_1}} R_{n_1}$

[WT] Well-trimmedness: $\forall a(n_1, n_2) \in A$. $R_{n_1} \in I_{t_{A(a)}}$

A partial configuration is finite if N is finite.

Figure 4 gives an intuition about the [DAC] requirement: if $R_{n_1} \subseteq_{J_{n_1 \to n_2}} R_{n_2}$, then each solution in R_{n_1} has a compatible solution in R_{n_2}, which in turn means that whatever the user will configure locally in the focus node f, will not cause inconsistency in any other node in the tree (DAC is transitive). MOVE and ASSIGN preserve properties of partial configurations, which means that DAC is an invariant of both algorithms. Exhaustive application of MOVE and ASSIGN reaches a particularly interesting class of *partial configurations*—directly corresponding to configurations of Def. 5:

DEFINITION 10. *A partial configuration is* final *if it is finite, all its R_n sets are singleton, and all the children of any node are instantiated: $\forall n_1 \in N. \forall e \in \operatorname{out} t_N(n_1)$. $R_{n_1} \subseteq I_e$ iff $\exists n_2 \in N$. $t_N(n_2) = \operatorname{dst} e$ and there is an arc $a(n_1, n_2) \in A$.*

Since for each vertex $S_v \subseteq \operatorname{Sol} P_v$, the only (and negligible) difference between final partial configurations and configurations is that the former store assignments in singleton sets (map R), while the latter store them directly (map s).

We state properties of our configurator analyzing the possible results of sequences of user actions consisting of ASSIGNs and MOVEs. All considered sequences begin in an initial partial configuration γ_0, which is created by calling INIT. The following theorem states that all reachable final partial configurations are correct configurations (i.e. there is no risk of misconfiguration):

THEOREM 11 (SOUNDNESS). *Any final partial configuration γ reachable from the initial partial configuration γ_0 by a sequence of calls to* MOVE *and* ASSIGN *applied to the initial partial configuration, is a configuration of the original modular configuration problem.*

Furthermore it is not possible to reach an inconsistent state:

THEOREM 12 (CONFLICT FREENESS). *Any partial configuration γ reachable from the initial partial configuration γ_0 by a sequence of calls to* MOVE *and* ASSIGN *contains no contradiction: for each node n in γ the set R_n is nonempty.*

This has a tremendously important usability implication: the configurator will never enforce back-tracking. The user can always continue, without getting stuck in the state, when no steps are allowed without violating constraints:

THEOREM 13 (BACKTRACK FREENESS). *In any non-final partial configuration γ it is possible to perform a finite sequence of* MOVE *operations after which either a final configuration is obtained, a call to* ASSIGN *is possible (the focus node is not finalized), or moves to not previously visited nodes are possible.*

But what if the only configurations reachable are infinite? Will the user be forced to continue forever? An attractive property of COMPILE is that it removes inherently infinite configurations, so they are never offered as a choice. The user can always be sure that he can complete his current configuration:

THEOREM 14 (TERMINABILITY). *From any non-final partial configuration γ it is possible to reach a final configuration γ' by applying a finite sequence of* MOVE *and* ASSIGN *operations. The configuration γ' obtained in that manner is guaranteed to be finite itself.*

Finally, our configurator is complete: it permits the construction of any finite configuration of a given problem:

THEOREM 15 (FINITARY COMPLETENESS). *Any finite configuration γ of a modular configuration problem M can be constructed by a finite number of* MOVE *and* ASSIGN *operations applied to the initial partial configuration γ_0.*

6. RELATED WORK & CONCLUSION

The success of decomposition techniques [6, 9] proves that configuration problems often contain an internal tree-like structure. Both [6, 9] try to discover and exploit this structure automatically in compilation-based configurators. Instead, we allow the designer to specify the structure explicitly, which gives us more precise information than in the decomposition approaches.

The unique feature of this work is the introduction of recursion, and thereby unboundedness, while retaining compilation and appealing theoretical properties that have a direct impact on user experience: soundness, completeness, conflict-freeness, backtrack-freeness, and terminability (protection against entering subspaces without finite final configurations). The response times are bounded with statically computable constants, which is crucial for user interaction.

To the best of our knowledge, no similar result is known for interactive CSP [8, 7, 4], where usually recursion is not supported, or systems work with backtrack search, or either conlict-freeness, completeness, or stringent run-time bounds are not provided.

The semantics of the configuration language used in the introduction has been studied in [1]. The implementation has been evaluated in an industrial case study, exhibiting imperceptible response times (31 modules and 247 variables at compile time, only counting shared variables once). Our configurator and the case study are described in [2].

7. REFERENCES

[1] E.R. van der Meer and H.R. Andersen. BDD-based recursive and conditional modular interactive product configuration. In *Proceedings of the International Workshop on CSP techniques with Immediate Application*, pages 112–126, Toronto, Canada, September 2004.

[2] Erik R. van der Meer. *Modular Configuration*. PhD thesis, ITU University of Copenhagen, 2005. Submitted.

[3] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[4] H. Fargier and M.-C. Vilarem. Compiling CSPs into tree-driven automata for interactive solving. In *Proceedings of the 3rd International Workshop on User-Interaction in Constraint Satisfaction*, pages 44–55, 2003.

[5] T. Hadzic, S. Subbarayan, R.M. Jensen, H.R. Andersen, J. Møller, and H. Hulgaard. Fast backtrack-free product configuration using a precompiled solution space representation. In *Proceedings of the International Conference on Economic, Technical and Organisational Aspects of Product Configuration Systems*, pages 131–138, Copenhagen, Denmark, June 2004.

[6] J. Nejsum Madsen. Methods for interactive constraint satisfaction. Master's thesis, University of Copenhagen, 2003.

[7] D. Magro. Interactive configuration capability in a sale support system: lazyness and focusing mechanisms. In *Proceedings of the Configuration Workshop held at IJCAI-2001*, pages 57–63, Seattle, Washington, August 2001.

[8] D. Sabin and E.C. Freuder. Configuration as composite constraint satisfaction. In *Proceedings of the 1st Artificial Intelligence and Manufacturing Research Planning Workshop*, pages 153–161, 1996.

[9] S. Subbarayan and H.R. Andersen. Linear functions for interactive configuration using join matching and CSP tree decomposition. In *Proceedings of the Configuration Workshop held at IJCAI-2005*, pages 7–12, Edinburgh, Scotland, July 2005.

[10] S. Subbarayan, R.M. Jensen, T. Hadzic, H.R. Andersen, H. Hulgaard, and J. Møller. Comparing two implemenations of a complete and backtrack-free interactive configurator. In *Proceedings of the International Workshop on CSP techniques with Immediate Application*, pages 97–111, Toronto, Canada, September 2004.