Compiling CSPs into tree-driven automata for interactive solving

Hélène Fargier[1] and Marie-Catherine Vilarem[2]

[1] IRIT, Toulouse
[2] LIRMM, Montpellier

Abstract. *Constraint programming techniques are widely used to model and solve decision problems and many algorithms have been developed to solve automatically and efficiently families of CSPs; nevertheless, they do not help solve interactive decision support problems, like product configuration. In such problems, the user chooses the values of the variables, and the role of the system is not to solve the CSP, but to help the user in this task. Dynamic global consistency maintaining is one of the most useful functionalities that should be offered by such a CSP platform. Unfortunately, this task is intractable in the worst case. Since interactivity requires short response times, intractability must be circumvented some way. To this end, compilation methods have been proposed that transform the original problem into a data structure allowing a short response time. In this paper, we extend the work of [1, 18] by the use of a new structure, tree-driven automata, that takes advantage of the structural characteristics of configuration problems (decomposition of the components into independent subcomponents). Tree-driven automata can be far more compact than classical automata while keeping their good properties, especially a tractable complexity for the maintenance of global consistency.*

1 Introduction and motivations

Constraint programming techniques are widely used to model and solve decision problems and many algorithms have been developed to solve automatically and efficiently some families of CSPs; nevertheless, they do not help solve decision support problems that are interactive in essence. For such problems, the user is in charge of the choice of values for the variables, and the role of the system is not to solve the CSP, but to help the user in this task. Product configuration [15][17] is a typical example of such problems: a configurable product is defined by a finite set of components, options, or more generally by a set of attributes, the values of which have to be chosen by the user. These values must satisfy a finite set of configuration constraints that encode the feasibility of the product, the compatibility between components, their availability, etc. A configurable product can thus be represented by means of a CSP, the solutions of which represents the catalog, i.e., all the feasible variants of the product.

When configuring a product, the user specifies the requirements by interactively giving values to variables or more generally by stating some unary constraints that restrict the possible values of the decision variables. Now, each time a new choice is made, the domains of the variables must be pruned so as to ensure that the values available for the further variables lead to a feasible product. These dynamical requests can also be removed during the configuration process because they eventually lead to a solution that is judged unacceptable by the user. Dynamic global consistency maintaining is thus one of the most useful functionalities that should be offered by the CSP platform.

Unfortunately, this task is intractable in the worst case [1]. Since interactivity requires short response times, intractability must be circumvented in some way. To this end, compilation methods have been proposed [1, 11, 18, 19], that transform the original problem into a data structure allowing a short response time. This principle was already be used in different domains of Artificial Intelligence, like planning under uncertainty [9] or automated reasoning [3] [4].

The approach of [1, 18] emphasizes the compilation of the original problem into a finite automaton, from which much better performances can be obtained: given a CSP $\Pi = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ and a permutation $O = [X_1, X_2, \ldots, X_n]$ of $\mathcal{X}$, each solution of $\Pi = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ defines a word of length n over the alphabet $\mathcal{D}$; hence, the set of solutions of Π defines a language over $\mathcal{D}$. This language, called the solution language of Π w.r.t. O a can be represented by a finite automaton (*FA*), i.e. a directed graph whose edges (the *transitions* of the automaton) are labelled by elements of the *alphabet*: a word $m = a_1 a_2 \ldots a_n$ is recognized by the automaton iff there exists a path from the initial state I to a final state F with the label m.

This approach takes advantage of the frequent interchangeability of values in real configurable products: such products can thus be described concisely by an automaton. But it does not handle the structural characteristics of the problem: complex products are generally structured into sub-components the existence of which depends on the values given to some of the variables of the upper component, but the sub-component of the same level are more or less independent from each other. This structural characteristics are better handled by (hyper)-trees of constraints and similar tree-like combinations of small (classical) CSPs: composite CSPs ([12][14]), cluster trees [11] and synthesis trees [19].

We thus aim at combining the compactness of the representation by automata with the principles of tree-like structures. The first idea should be to use the constraint (hyper)-tree to derive an efficient order O on which an automaton shall be build. However, a preliminary study [2] has shown that finding the order minimizing the size of the automaton is an intractable problem, and worst, experiments on some tree-structured problems suggests that some of them lead to an exponentially-sized automaton, whatever O. We will show in the following that this result is confirmed by a proof of spatial complexity.

So, we shall drop the compilation linear automata as such. In the present paper we rather propose generalization of this structure that addresses the former tree-like characteristics: the tree-driven automaton. Briefly, it can be understood as a tree of small linear automata, each of them corresponding to a sub-component (and recursively). Importantly, it can be far more compact than the classical automaton (in particular any acyclic CSP yields a polynomially sized tree-driven automaton) while keeping its good properties for the maintenance of global consistency, especially a tractable time complexity.

In Section 2, we recall how a linear automaton can represent the set of solutions of a CSP and show that the restriction to trees of constraints is still intractable. Section 3 then presents our basic definition on tree-driven automata and Section 4 then briefly shows how the task maintaining global consistency can be efficiently achieved on this structure. Section 5 details how to compile a CSP into a tree-driven automaton, and shows that this can be done in polytime for acyclic problems.

2 Compiling a CSP into a finite automaton

2.1 Definitions [18]

A finite-state automaton (FA, for short) is a directed graph; its edges (the *transitions* of the automaton) are labelled by elements of a set Σ (the *alphabet*); its vertices are called *states*; this graph has one initial state I (usually, the root of the graph) and at least one final state. We consider here automata such that any transition and any state belong to at least one path from I to a final state[3]. The finite automaton is said to be deterministic if and only if all the transitions coming from a single state have different labels – determinism ensures the uniqueness of the representation, but is not necessary for the purpose of our study. A word $m = a_1a_2 \dots a_n$ is recognized by the automaton if there exists a path from the initial state I to a final state F with the label m. The language recognized by the automaton is the set of words it recognizes.

Let $\Pi = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a CSP. Given a permutation $O = [X_1, X_2, \dots, X_n]$ of $\mathcal{X}$ called the *support* of the automaton, any solution of $\Pi = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ defines a word of length n over the alphabet $\mathcal{D}$. Hence, the set of solutions of Π defines a language over $\mathcal{D}$. This language, called the solution language of Π w.r.t. O and denoted $\mathcal{S}_O$ can be represented by a FA. This FA has only one initial state I and one final state F, and is *uniform*, i.e. the length of any path from I to F is n. So, its states can be clustered into *levels* (the rank of the state in the graph defines its level) such that the transitions always map one state of one level to one state of the adjacent level.

Example 1. Let $\Pi = (\{X_1, X_2, X_3\}, \{a, b, c\}, \{C_1, C_2, C_3\})$ where C_1 is the constraint $X_1 \neq X_2$, C_2 is $X_2 \geq X_3$ and C_3 is $X_1 \neq X_3$. Figure 1 shows the automaton associated to Π, given the support $O = [X_1, X_2, X_3]$.

2.2 Basic operations and building

In theory, the automaton representing a CSP could be build from the extensive list of its solutions – this is hopefully not the case in practice. Following [18][1], it is rather obtained by the composition of (small) automata representing the

[3] In automata theory, they are called "trimmed" automata

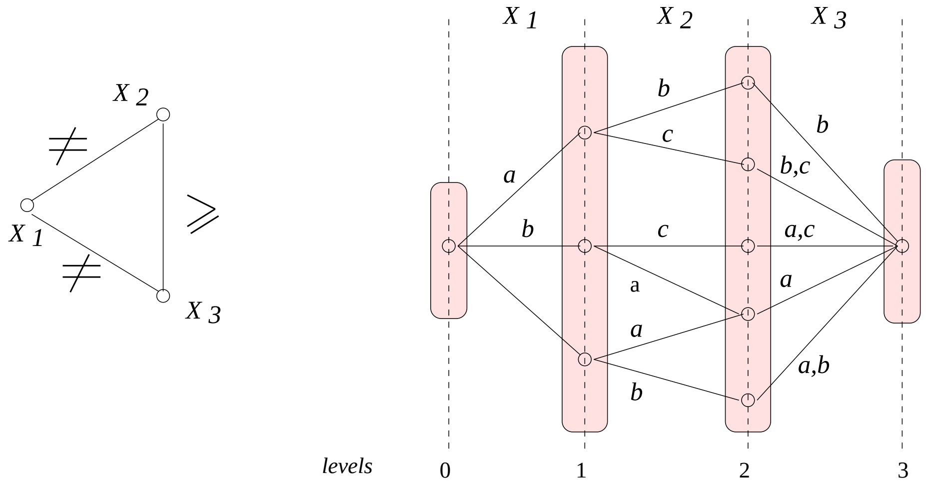

Fig. 1. The CSP Π is on the left, and its associated automaton with support $[X_1, X_2, X_3]$ is in the right

constraints of $\mathcal{C}$. The main automata operations used in this kind of process are the *disjunction*, the *conjunction* and the *extension*, together with three auxiliary operations: the *fusion of the equivalent states*, the *determinization*, and the *trimming* of the automaton. Let us sketch the first ones (see [2] for more details the operations on CSP-automata):

Definition 1. *Let $\mathcal{A}$ and $\mathcal{A}'$ be two automata sharing the same support*

The disjunction *of $\mathcal{A}$ and $\mathcal{A}'$ is the automaton which recognizes the union of the languages recognized respectively by $\mathcal{A}$ and $\mathcal{A}'$; it will represent the solutions of $\Pi \vee \Pi'$.*

The conjunction *(or* composition*) of $\mathcal{A}$ and $\mathcal{A}'$ is the automaton which recognizes the intersection of the languages recognized respectively by $\mathcal{A}$ and $\mathcal{A}'$; it will represent the solutions of $\Pi \wedge \Pi'$.*

Both operations are performed through the construction of a third automaton, the *product automaton*, which simulates the working of both $\mathcal{A}$ and $\mathcal{A}'$: each state of the product automaton is a pair (q, q') where q (resp. q') is a state of $\mathcal{A}$ (resp. $\mathcal{A}'$). There is a transition labelled by a between (q_1, q_1') and (q_2, q_2') if and only if (i) there is in $\mathcal{A}$ a transition labelled by a between q_1 and q_2 and (ii) there is in $\mathcal{A}'$ a transition labelled by a between q_1' and q_2'. By using an appropriate selection of the final states of the product automaton, we get either the disjunction or the conjunction of the two input automata.

The worst time complexity of both constructions is rather high in the general case of automata theory, as well as the size of the product automaton; fortunately, it is less costly in the case of CSP-automata, since they are uniform: the construction can be done level by level since only the states (q, q') where q

and q' are on the same level are to be built. The time complexity can thud be reduced to $O(n \times |\mathcal{D}| \times width(\mathcal{A}) \times width(\mathcal{A}'))$, where the width of an automaton is the maximum number of states on a level.

In order two soundly achieve the union (or the intersection) of solution sets, the previous operations require automata sharing the same support (and hence CSPs sharing the same set of variables). If it is not the case, one may extend both to a common support. Let $\mathcal{A}$ be the automaton of some CSP Π with variables $\mathcal{X}$ and $O = [X_1, X_2, \ldots, X_n]$ its support. The extension of Π to a new variable $X \notin \mathcal{X}$ is obviously the CSP Π' with variables $\mathcal{X} \cup \{X\}$ and with the same constraints as Π. O is extended to X by the insertion of X at some given position k — this lead to a support $O' = [X_1, \ldots, X_{k-1}, x, X_{k+1}, \ldots, X_n]$. The solution language of Π' the one recognized by the extension of $\mathcal{A}$ to $O' = [X_1, \ldots, X_{k-1}, X, X_{k+1}, \ldots, X_n]$, which is an automaton $\mathcal{A}'$ with support O'. In order to build $\mathcal{A}'$, we need only to make a copy of level $k-1$, to insert this copy between levels $k-1$ and k, and, for every state q of level $k-1$, to add a transition labelled by $\mathcal{D}$ to the copy of q (see Figure 2). The time complexity of this operation is $O(width(\mathcal{A}) \times |\mathcal{D}|)$.

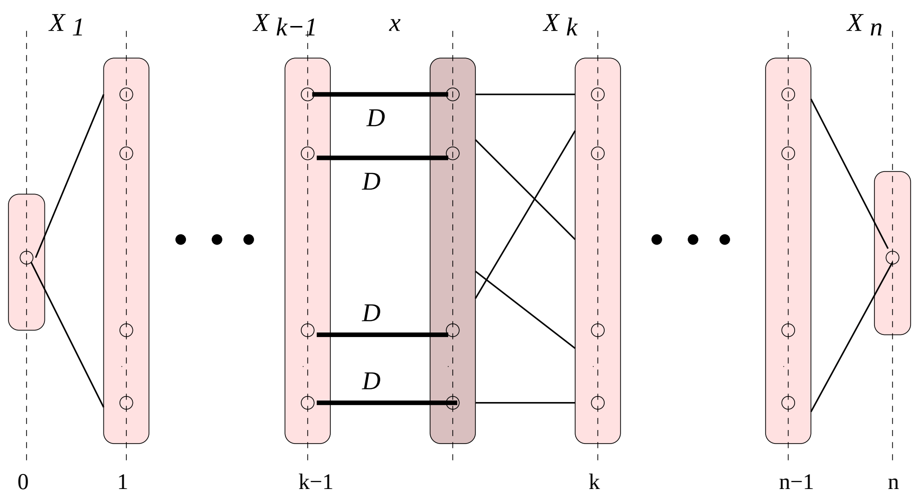

Fig. 2. The darker level is the copy of level $k-1$ and the thick edges correspond to the new transitions; they are labelled by $\mathcal{D}$.

The compilation of a CSP is thus achieved in the following simple way: first, one build the automaton associated to each constraint (using the disjunction operator). These small automata are then composed through the extension and conjunction operations. Operators of trimming, fusion of equivalent states and/or determinisation can be applied at each step, e.g. for removing redundant and/or useless states (again, see [2] for more details and experiments).

2.3 Complexity results

The complexity of the computation obviously depends on the size of the automaton, which is exponential in the worst case. In practice, this size is mainly influenced by the order in which the variables of the CSP are taken into account, but it has unfortunately be shown that the determination of the order minimizing the size is an NP-hard problem [2]. Experiments drawn by the same author suggest that the theoretical size of the automaton remain exponential *even when the graph of constraint is a tree.* The first important result of the present paper is the formal proof of this conjecture.

Proposition 1. *There exists tree-structured CSPs which don't have a polynomial size representation by a classical (linear) automaton.*

The proof, which can be found in the appendix, is supported by the following sub-class of constraint trees. It can indeed be shown that, whatever the order $\mathcal{O}$ chosen, the compilation of any instance of this class leads to a non-polynomial linear automaton:

Definition 2. *Let h, k, d be three positive integers, with $k \geq 3, k < d-1$. The CSP $\Pi(h,k,d) = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ is defined by:*

- *the constraint-graph is the complete k-ary tree with height h (see Figure 3 for an example with $k = 3$ and $h = 2$)*
  Therefore, we have $n = |\mathcal{X}| = \Sigma_{i=1,\ldots,h}\ k^i = (k^{h+1}-1)/(k-1)$.
- *$D_1 = \ldots = D_n = D = \{1, \ldots, d\}$*
- *each constraint relation is the relation $\neq$*

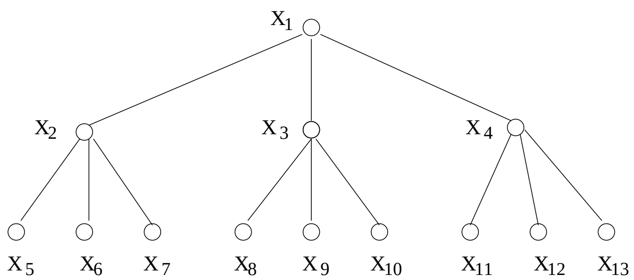

Fig. 3. The constraint graph of $\Pi(h,k,d)$ where $h = 2$ and $k = 3$.

So, the tractability of tree-structured of CSP, and more generally of hyper trees of constraints, is lost upon its compilation by a linear automata. This theoretical result has severe consequences on our practical problems of configuration: one of their main characteristics is precisely their tree-like structure (each hyper-edge corresponding to a sub-component).

We are thus facing the following dilemma: either we compile configuration problems into automata, but we lost the advantages due to their structuration into hyper-trees of components, or we choose an approach by tree clustering ([5]; see [11] for the application of tree clustering to dynamical configuration problems), but in this case we should get a data-structure exponential in the size of the subcomponent (each sub-component leading to an hyper-edge) and we loss the the advantages due to the interchangeability of values and of components.

The main idea of our work is then to use a tree of small automata, to compile an automata for each hyper-edge and to maintain the interface between them, or, equivalently to build tree-driven automata.

3 Tree-driven automata

3.1 Definitions

Shortly, a tree-driven automaton is an automaton whose support is not a chain, but a tree.

Definition 3. *Let (Q, E) be a graph – Q being the set of its vertices and E its set of edges.*

A coloring *τ of the vertices assigns to each vertex a number, such that two adjacent vertices have different colors.*

The quotient *of a graph (Q, E) by a coloring τ of its vertices, is the graph $\mathcal{T} = (Q_\mathcal{T}, E_\mathcal{T})$ defined by:*

$$Q_\mathcal{T} = \{\tau(q), q \in Q\} = \{0, 1, \dots, n\}$$
$$E_\mathcal{T} = \{\{i, j\} / \exists\ q, q' \in Q \ \text{s.t.}\ \tau(q) = i, \tau(q') = j, \{q, q'\} \in E\}$$

Definition 4. *A* tree-driven automaton *$\mathcal{A}$ of order n on the alphabet Σ is a graph (Q, E) such that:*

- *there is a coloring of the vertices Q with $n + 1$ colors $(0, \dots, n)$, denoted by τ: each vertex is labelled by an element of $\{0, 1, \dots, n\}$ and adjacent vertices have different colors*
- *the edges are labelled by subsets of Σ; this labelling is denoted by σ*
- *the quotient (Q, E) by τ, denoted by $\mathcal{T} = (Q_\mathcal{T}, E_\mathcal{T})$ is a tree called the* support *of $\mathcal{A}$.*

A transition *of $\mathcal{A}$ is a pair $\langle a, e\rangle$ where e is an edge of E and a an element of Σ belonging to the labelling of e ($a \in \sigma(e)$).*

The vertices of $\mathcal{A}$ are called states, *as we do for linear automata.*

In other words, τ clusters the states (and thus the edges) of the automaton into disjoints levels organized in a tree — the support of $\mathcal{A}$. An example is given in Figure 3.1. Q_i will denote the set of nodes of level i and $E_{i,j}$ the set of transitions that join level i to level j (the transitions of level i, j, for short) :

$$Q_i = \{q \in Q, \tau(q) = i\}, i \in Q_\mathcal{T} = \{0, 1, \dots, n\}$$

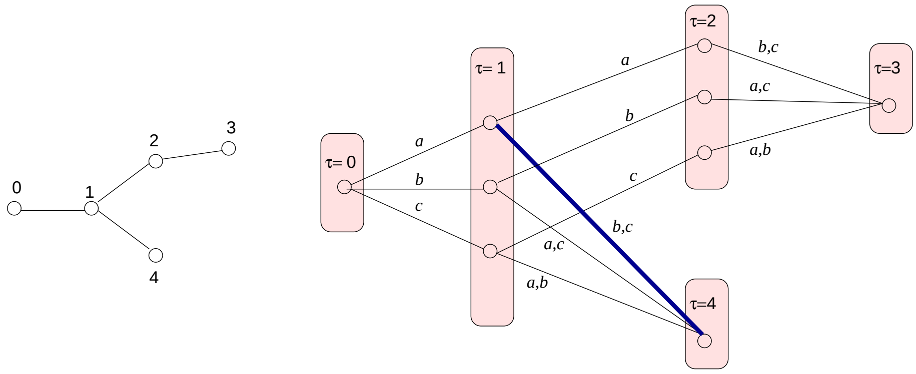

Fig. 4. A tree-driven automaton and its support (on the left)

.

$$E_{i,j} = \{(q, q') \in E, \tau(q) = i, \tau(q') = j\}, \{i, j\} \in E_{\mathcal{T}}$$

Since a path is a tree, a classical (linear) automaton is no more than a tree-driven automaton the support of which is a chain (it is enough to build a τ that maps every vertex of rank X_i to i.

Definition 5.

A word *on a support $\mathcal{T} = (Q_{\mathcal{T}}, E_{\mathcal{T}})$ is a labelling w of $E_{\mathcal{T}}$ which associates to each edge of $\mathcal{T}$ an element of Σ.*

A $\mathcal{T}$-tree *t is a labelled subgraph (Q_t, E_t, w_t) of $\mathcal{A}$ which is isomorphic to $\mathcal{T}$ (i.e. (Q_t, E_t) is a tree, τ induces a one-to-one correspondence between Q_t and $Q_{\mathcal{T}}$ and between E_t and $E_{\mathcal{T}}$) and, for each edge e of $E_t, w_t(e) \in \sigma(e)$.*

Definition 6.

A word w is recognized *by a tree-driven automaton $\mathcal{A}$ with support $\mathcal{T}$ if it has an associated $\mathcal{T}$-tree in $\mathcal{A}$, i.e. if there exists a $\mathcal{T}$-tree (Q_t, E_t, w_t) of $\mathcal{A}$ such that for each e in E_t, $w_t(e) = w(\tau(e))$.*

The language *recognized by a tree-driven automaton $\mathcal{A}$ is the set $\mathcal{L}(\mathcal{A})$ of the words that it recognizes.*

Two tree-driven automata built on the same support are said equivalent *iff they recognize the same language.*

Notice that in this definitions all the words are supposed to share the same support. An example is given in Figure 5. A more sophisticated generalization of automata can be proposed, so as to allow words of different lengths, but this is out of the scope of this paper, that focus on the generalization to non linear structures of uniform automata.

Now, given a suitable support, i.e. a graph whose edges are in one-to-one correspondence with the variables of $\mathcal{X}$, each solution of a CSP can be identified with a word. The idea is to exploit a tree-driven automaton the language of

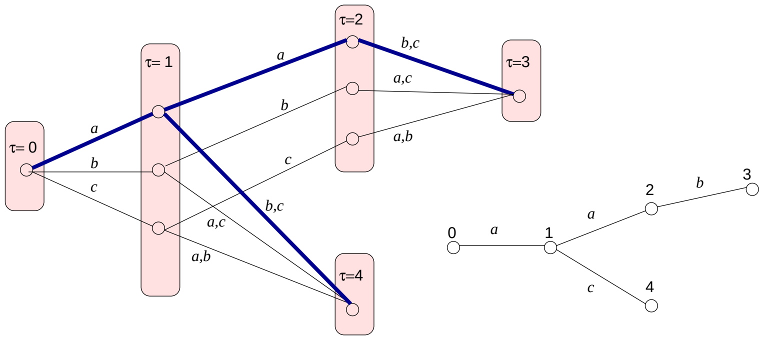

Fig. 5. An example of a word recognized by the tree-driven automaton on the left; the bold edges of the tree-driven automaton define four $\mathcal{T}$-tree , one of them corresponds to the word on the right

which is precisely this set of words. The next Section shows how to use it for the task under concerns, namely the dynamical maintaining of global consistency and Section 5 explains how to build it in practice — we will see that for any tree of constraints, a tree-driven automaton can be build, the size of which is linear w.r.t. the size of the micro structure of the CSP. But let us first conclude our definition of tree-driven automata by the extension of the basic operations on linear uniform automata.

3.2 Basic operations on tree-driven automata

Normalization Let $\mathcal{A} = (Q, E)$ be a tree-driven automaton with support $\mathcal{T}$ and let $Q_i = \{q \in Q$ such that $\tau(q) = i\}$. When i is a leaf of $\mathcal{T}$, it is easy to see that collapsing all the vertices of Q_i leads to an automaton equivalent to $\mathcal{A}$. This operation is called the normalization of the automaton and can be achieved in linear time in the size of (Q, E). It lead to a normalized one, i.e. an automaton such that for each leaf i of $\mathcal{T}$, $\#Q_i = 1$..

Trimming A tree-driven automaton $\mathcal{A}$ may contain useless vertices: vertices which do not belong to any subtree isomorphic to the support. Formally, a vertex $q \in Q_i$ of $\mathcal{A} = (Q, E)$ is useless if $\exists j$ s.t. $\{i, j\} \in E_{\mathcal{T}}$, and there is no edge between q and a vertex of Q_j.

The suppression of the useless vertices (and of their incident edges) yields an equivalent automaton and can be achieved in linear time in the size of (Q, E).

In the following, we consider only normalized and trimmed automata.

Determinization of tree-driven automata It is also possible to extend the classical definition and properties of deterministic automata to tree-driven automata. The key point is that the support is no longer an undirected tree, but must be a directed rooted tree (so, one consider that τ defines a directed rooted tree): for any state q, $\tau(q)$ is called the level of q.

A tree driven automaton is said to be deterministic if and only if it is normalized, it only has one initial state (of level 0), and for any couple of transitions (x_1, y_1) and (x_2, y_2) of the same level ($\tau(x_1) = \tau(x_2)$ and $\tau(y_1) = \tau(y_2)$), $x_1 = x_2$ and $y_1 \neq y_2 \Rightarrow \sigma((x_1, y_1)) \cap \sigma((x_2, y_2)) = \emptyset$. The definition is quasi-equivalent to the one provided for linear automata, and it can be proved that, like in the latter case, the deterministic representation of a language is unique (up to a given τ).

Any tree driven automaton can be determinized with a procedure very close to the one used for classical automata. As it is the case for linear automata, this operation may be exponential in time, and the deterministic tree automata can be exponentially larger that the original one (this is easily explainable noticing that any linear automata is a tree-automata). In the sole context on interactive CSP solving, determinism is not compulsory.

Fusion of equivalent states. Two states q and q' belonging to the same level i are *equivalent* if, for all neighbors j of i in the support $\mathcal{T}$ but one denoted p, for any $q_j \in Q_j$, we have: $\langle a, (q, q_j)\rangle$ iff $\langle a, (q', q_j)\rangle$. By merging q and q', we get an automaton equivalent to $\mathcal{A}$. This operation reduces the size of the automaton, and can be done in polynomial time.

Automata composition. The operation of composition of automata (or "conjunction") can be straightforwardly extended to automata defined on the same support. Let $\mathcal{A}_1 = (Q_1, E_1, \tau_1, \sigma_1)$ and $\mathcal{A}_2 = (Q_2, E_2, \tau_2, \sigma_2)$ share the same support $\mathcal{T}$. The product $\mathcal{A}_1 \times \mathcal{A}_2 = (Q, E, \tau, \sigma, \Sigma)$, is a tree-driven automata with support $\mathcal{T}$ such that $Lang(\mathcal{A}_1 \times \mathcal{A}_2) = Lang(\mathcal{A}_1) \bigcap Lang(\mathcal{A}_1)$. It is defined by:

- $Q = \bigcup_{i \in Q_{\mathcal{T}}} (Q_{1_i} \times Q_{2_i})$
- $E = \{\ \{(x_1, x_2), (y_1, y_2)\} / \{x_1, y_1\} \in E_1, \{x_2, y_2\} \in E_2$ and $\sigma(\{x_1, y_1\}) \cap \sigma(\{x_2, y_2\}) \neq \emptyset\ \}$
- $\forall x_1 \in Q_{1_i}, x_2 \in Q_{2_i}, \tau((x_1, x_2)) = \tau_1(x_1) = \tau_2(x_2) = i$
- $\forall \{(x_1, x_2), (y_1, y_2)\} \in E, \sigma(\{(x_1, x_2), (y_1, y_2)\}) = \sigma_1(\{x_1, y_1\}) \cap \sigma_2(\{x_2, y_2\})$

The complexity of this operation is not worse than in the linear case, but stays obviously in the same class of complexity (again, since any linear automata is a tree-driven automata). Notice that determinism is conserved upon the operation of conjunctive composition. A step of fusion of the equivalent states shall be applied on the resulting product automaton.

The operation of disjunction of automata can also be defined (the disjunction $\mathcal{A}_1 \vee \mathcal{A}_2 = (Q, E, \tau, \sigma, \Sigma)$ is the tree-driven automata on the same support such that $Lang(\mathcal{A}_1 \vee \mathcal{A}_2) = Lang(\mathcal{A}_1) \bigcup Lang(\mathcal{A}_1)$.) but is of no use in the scope of the present paper.

4 Maintaining global consistency

The practical compilation of a CSP into a tree driven automaton is detailed in Section 5. For the moment, assume that a tree-driven automaton has been obtained from a CSP $\Pi = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ — it represents for instance all the feasible assignments of a configurable product. A set $\mathcal{H}$ of constraints represents the current set of user's choices; like in [1] we assume that $\mathcal{H}$ is a set of unary constraints. The decision support system should at least achieve the following tasks:

(1) detection of inconsistency of $\Pi' = (\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \mathcal{H})$: at any time, the system must tell whether there is a feasible product satisfying the user's requirements
(2) maintenance of global consistency: at any time, the system must discard from the domain of available values those that cannot lead to a feasible product, i.e. it must be able to compute the projection of the solution set of Π' on each variable X_i.

The approach used to efficiently perform these tasks on a tree-driven automaton is a straightforward extension of the approach used for the linear case [1]. The principle is to associate a cost ϕ to each transition of the automaton: for a transition $\langle a, e\rangle$of level i, the weight $\phi(\langle a, e\rangle)$ is the number of constraints of $\mathcal{H}$ that explicitly forbid the assignment $X_i = e$ – it is equal to 0 it the assignment is allowed. For sake of simplicity, we thus assume here that all the constraint have the same weight 1; it is easy to extend this by associating a weight to any constraint of $\mathcal{H}$. The global cost of a $\mathcal{T}$-tree (resp. a state, a transition) can now be defined:

Definition 7. *Let $\mathcal{A} = (Q, E)$ be a tree-driven automaton with support $\mathcal{T}$.*

- *cost(t)* $= \Sigma_{\langle a,e\rangle \in t}\ \phi(e, a)$ *is the* cost of a *$\mathcal{T}$-tree t.*
- *cost($\langle a, e\rangle$) is the cost the best (i.e. minimal cost) $\mathcal{T}$-tree containing $\langle a, e\rangle$*
- *cost(q) with $q \in Q$ is the cost of a best $\mathcal{T}$-tree containing q*

The following propositions can then be proved (their proofs are straightforward extensions of those of [1] and are omitted for the shake of brevity).

Proposition 2. *The CSP $\Pi' = (\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \mathcal{H})$ is consistent if and only if there is in $\mathcal{A}$ a $\mathcal{T}$-tree of cost 0 (or equivalently if cost(q_0) $= 0$)*

Proposition 3. *Let P_{X_i} denote the projection of the solution set $S(\Pi')$ on variable X_i. Then, e belongs to P_{X_i} if and only if there is a transition a of level $(pred(i), i)$ in $\mathcal{A}$ such that $e \in \sigma(a)$ and $\langle a, e\rangle$ belongs to a $\mathcal{T}$-tree of cost 0.*

Tasks (1) and (2) thus amount to the search of $\mathcal{T}$-tree of objects minimal cost in the tree-driven automaton. Two allow an efficient computation of the costs, we direct the support and thus the automaton (e.g. choosing level 0 for root and thus q_0 for initial state) and we associate to each state q two counters: $c_l(q)$ and $c_r(q)$.

Definition 8.

- *for the unique* $q_0 \in Q_0$, $c_l(q) = 0$
- *for each* $i \in Q_{\mathcal{T}}$ *which is a leaf,* $\forall q \in Q_i, c_r(q) = 0$
- *for* $q \in Q_i$,
  - $c_l(q) = \sum_{j \in Pred(i)} Min_{a,out(a)=q}\ (c_l[q] + \phi(a))$
  - $c_r(q) = \sum_{j \in Succ(i)} Min_{a,in(a)=q}\ (c_r[q] + \phi(a))$

  where $Pred(i)$ *(resp.* $Succ(i)$*) is the set of predecessors (resp. successors) of* i *in the support.*

For any state q of $Q_{\mathcal{T}}$ which is not a leaf, q induces two connected components; by convention we call the one containing the state q_0 the *left* sub-automaton of q and the other its *right* sub-automaton. It is easy to show that $c_l(q)$ (resp. $c_r(q)$) is the cost of the best $\mathcal{T}$-tree in the sub-automaton left (resp. right) sub-automaton of q. Finally, Bellman's principle is obviously satisfied and the cost of the best $\mathcal{T}$-tree through a state q or a transition $\langle a, e\rangle$ can therefore be directly deduced from these the left and right costs:

Proposition 4.

- $cost(q) = c_r(q) + c_r(q)$.;
- $cost(\langle a, e\rangle) = c_l[in(\langle a, e\rangle)] + \phi(\langle a, e\rangle) + c_r[out(\langle a, e\rangle)]$.

At the initialization phase, all the costs are obviously set to 0 (no value has still been deleted).

The interactive configuration process then goes on, though the addition or the retraction of unary constraints. Notice that these unary constraints can affect any of the variables. A strict partial order is necessary for building the automata, but **not** for decision-making (it is even possible to add several constraints on the same variable at different instants of the configuration process). Technically, when a unary constraint is added or deleted the costs of some transitions may be modified (incremented in the first case, decremented in the second one). The left and right counters then be maintained by propagation algorithms which are very similar to those used for linear automata: the modification of the cost of a transition has to be propagated backward (to update c_r) and forward (to update c_l):
1) Backward propagation from the left state of each of these transitions: this is done through a breadth-first search strategy, so as to update the right costs c_r of the traversed states knowing the right costs of their successors.
2) Forward propagation from the rigth states of these transitions: it is also done with a breadth-first search strategy, so as to update the left costs c_l of the traversed states knowing the left costs of their predecessors. This work can be done in time polynomial in the size of the tree-driven automaton.

Practically, counters $cnt()$ can be attached to the pairs (i, v) recording the number of transitions of level i that are labelled by the value v (the number of transition supporting $X_i = v$). They are easily maintained during propagations: for any transition $\langle a, e\rangle$ of level i encountered, $cnt(i, v)$ is incremented if the cost of the transition rises from 0 to a positive value ($\langle a, e\rangle$is no more one of the transition of cost 0 that supports the assignment $X_i = e$) and conversely, it is decremented if its comes down to 0. Any value without support ($cnt(i, v) = 0$) is withdrawn from its domain. Upon the relaxation of some constraint any value which recover a first support it re-introduced into its domain.

5 Building tree-driven automata: space complexity and algorithms

The tractability of our approach obviously depends on the ability to build the original automaton and then of its size — the previous algorithms being polynomial with respect to this size. Tree driven automata are obviously not built from the extensive list of all the solution of the CSP: to be useful, the approach should minimize not only the size of the final automaton, but, as soon as possible, restrict the space at each step of the building. The choice of the support is particularly crucial and should minimize the size of the automaton. Moreover, recall that we aim at proving that tree driven automata are efficient on tree-structured problems, i.e. that this size in bounded for this class of CSP.

The first paragraphs of this section will deal with the kind of CSP we are focusing on, namely trees of constraints (paragraph 5.1), or more generally hypertrees of constraints (paragraph 5.2)– the general case being sketched in paragraph 5.3

5.1 Tree-driven automata for tree-structured CSPs

When the network of constraints is tree-structured, a tree-driven automaton can be built which is very close to its micro-structure. The support $\mathcal{T} = (Q_{\mathcal{T}}, E_{\mathcal{T}})$ is build as follows: we can suppose without loss of generality then the constraint tree is rooted and the variables numbered from 1 to n according to this orientation. $Q_{\mathcal{T}}$ is taken equal to $\{1, \ldots, n\} \cup \{O\}$ — vertex is called the *handle* and will roots the tree) Two vertices i and j of $Q_{\mathcal{T}}$ are linked by an edge iff there is in the constraint tree a constraint between X_i and X_j. Vertex 0 is linked vertex 1. This new graph is the support of the forthcoming automaton. In particular, each edge (i, j) such that $i < j$ of $E_{\mathcal{T}}$ can be mapped to X_j: this defines a one-to-one correspondence between $\mathcal{X}$ and the edges of the support .

The vertices (resp. the edges) of the micro structure will constitute the set of state (resp. edges) of the automaton : the vertices q corresponding to D_i will receive the label $\tau(q) = i$. $\mathcal{A}$ is completed by the addition a new vertex q_0 which is linked to all the vertices corresponding to D_1. q_0 is the root of a tree and receive label $\tau(q_0) = 0$. The labelling σ is then obtained by making the values (that originally pertain to the vertices of the micro structure) "slide" along the incident edges. The operations of trimming, normalization, fusion of equivalent notes and, if needed, determinization can then be applied. Notice that the trimming of the automaton is equivalent to a stage of arc-consistency on the micro-structure. The fusion of equivalent nodes corresponds to a simplification by interchangeability.

Example 2. The tree-driven automaton of Figure 5 is the one built by this procedure from the micro-structure of the tree of constraint given by Figure 6. Figure 7 shows the one-to-one correspondence between the variables of the CSP and the edges of the support.

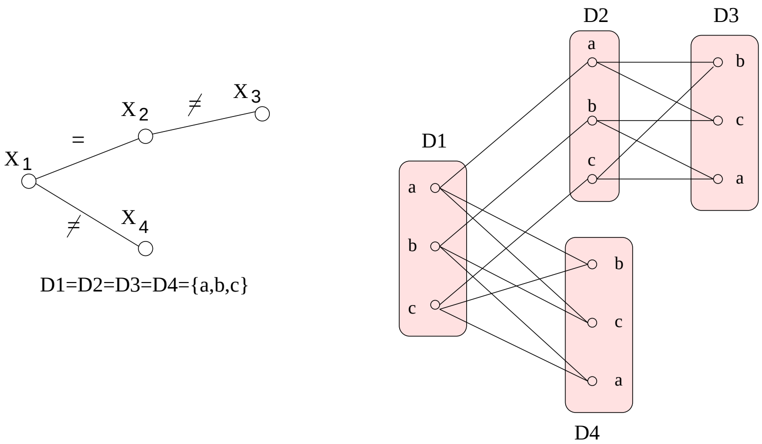

Fig. 6. A CSP Π and its micro-structure

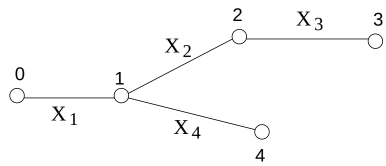

Fig. 7. The support computed of the CSP of Figure 6

Importantly, this algorithm build an automaton the size of which is given by the size of its micro-structure.

Proposition 5. *Let Π be a tree-structured CSP, and let $\mathcal{A}$ be the associated normalized tree-driven automaton constructed as in Section 5.1. The size of $\mathcal{A}$ is linear in $size(\Pi)$, by the size of its micro-structure of Π*

This is trivial when considering the previous construction

Hence, contrarily to linear automata theory, tree-driven automata theory do allow a polynomial representation of any tree of constraints. This complexity

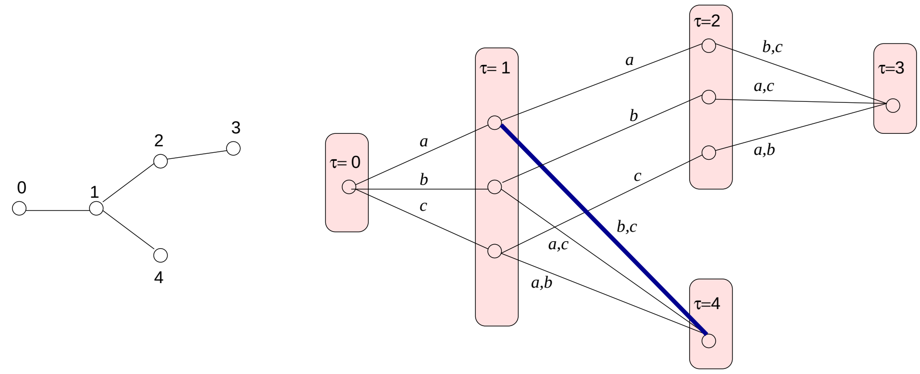

Fig. 8. The tree driven automaton built form the CSP of Figure 6 in accordance with the support of Figure 7

results constitutes the ground theoretical argument in favor of the use of tree-driven automata for represneting configuration problems.

5.2 Tree-driven automata for hyper-trees of constraints

As outlined by [11][19], the shell of many configuration problems, and namely of Vehicle configuration problems is an hyper-tree of constraints, i.e. an acyclic hypergraph. The idea, when compiling hyper-trees of constraints, is to compute one small, linear, automata for each constraint, and then to combine them into a tree automata. Let us first build the support of the automaton. Since the hypergraph is acyclic, it is always possible to build a directed tree on $\mathcal{X}$ that satisfies the following property **P**:

P: For each constraint C_i, the variables of C_i are on a same path of G

This directed tree defines a partial order $\mathcal{O}$ ($x < y$ iff there is a path from x to y) having a least element (the root). In the following, we use $\mathcal{O}$ to denote as well the directed tree as the induced order. Property **P** ensures that the variables of any C_i are totally ordered by $\mathcal{O}$ and therefore each constraint has a greatest variable w.r.t the order $\mathcal{O}$.

Example 3. Let us consider the hypertree on $\mathcal{X} = \{X_1, \ldots, X_{15}\}$ defined by the following seven constraints:
$C_1 = \{2, 3, 6, 7\}$ $C_2 = \{1, 2, 3, 4, 5\}$ $C_3 = \{2, 5, 12\}$
$C_4 = \{4, 5, 11\}$ $C_5 = \{3, 4, 10\}$ $C_6 = \{3, 7, 8, 9\}$ $C_7 = \{11, 13, 14, 15\}$
Figure 9 shows the corresponding directed tree. It is easy to check that **P** is satisfied.

A "handle" (0) is then added to $\mathcal{O}$ as well as an arc from 0 to the root of the directed tree. We thus obtain a tree with vertices $\{0, \ldots, n\}$ – it is the support $\mathcal{T}$

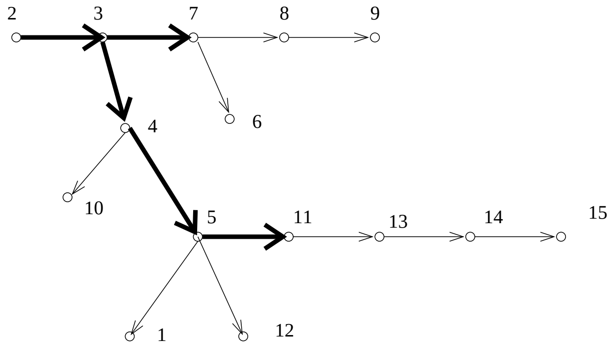

Fig. 9. A shell of a support for the hypertree of constraints of example 3

of the forthcoming automaton. The reader can check that there is a one-to-one correspondence between the edges of this tree and the variables of the CSP.

Algorithm 1 shows how to build a directed tree satisfying **P**. This algorithm does not input the hypertree but a *join tree* $\mathcal{JT}(\mathcal{C}, E)$ (or *junction tree* using the vocabulary of Bayesian nets): a joint graph for a CSP is graph whose vertices are the constraints of $\mathcal{C}$ and such that, whenever the same variable X_i occurs in two constraints C_1 and C_2, then C_1 and C_2 are connected in the tree and X_i occurs on each vertex on the paths linking C_1 and C_2 in the join graph. When the join graph is a join tree, this path is unique. This is done without loss of generality, since it is well known that an hypergraph is acyclic if and only it has a join tree, which can be computed in polytime (see e.g. **??**, **??**). For instance, Figure10 shows the join tree for the CSP of example 3.

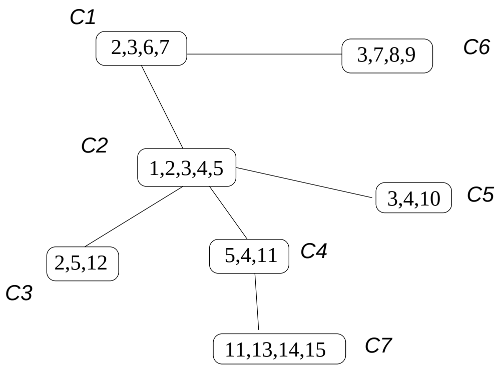

Fig. 10. A join tree for the CSP of example 3

The algorithm works in two steps: the first step builds a directed tree on the variables belonging to at least two constraints (this directed tree is pictured by the bold arcs of Figure9); then, the second step adds the remaining variables. The complexity is linear w.r.t. the size of the join tree.

Algorithm 1: Construction of an directed tree on $\mathcal{X}$ which induces a total order $\mathcal{O}$ on each C_i of $\mathcal{C}$

Data: The join tree $\mathcal{JT}$ and C_i a vertex of $\mathcal{JT}$
Result: The directed tree denoted $\mathcal{O} = (\mathcal{X}, U)$

```
M ← ∅ ;
U ← ∅ ;
O ← (M, U) ;
/* step 1: constructing the arborescence on the variables belonging to at
least two constraints */
Depth-first-search(C1)
/* ∀Cj ∈ C, the variables of Cj ∩ M are on a same path of O */
/* step 2: adding the remaining variables to O */
for all Ci ∈ C do
    Xm ← the variable of Ci ∩ M which is the greatest for O;
    R ← Ci \ M;
    /* Let R = {Xi,1, Xi,2, ..., Xi,r} */
    /* Adding the path Xm, Xi,1, Xi,2, ..., Xi,r to O */
    M ← M ∪ R ;
    for k from 1 to r-1 do
        U ← U ∪ { (Xi,k, Xi,k+1 )};
    end
    U ← U ∪ { (Xm < Xi,1 )};
end
```

Procedure `Depth-first-search`(C_i)

Data: The join tree $\mathcal{JT}$ and C_i a vertex of $\mathcal{JT}$

Result: Add to $\mathcal{O}$ the variables of C_i not yet belonging to $\mathcal{O}$

$I \leftarrow (\bigcup_{\{C_i,C_j\}\in E} C_i \cap C_j) \setminus M$;

/* *Let* $I = \{X_{i,1}, X_{i,2}, \ldots, X_{i,r}\}$ */

if $I \neq \emptyset$ **then**

  $X_m \leftarrow$ the variable of $C_i \cap M$ which is the greatest for $\mathcal{O}$;

  /* adding the variables of I */

  $M \leftarrow M \cup I$;

  for *k from 1 to r-1* **do**

    $U \leftarrow U \cup \{(X_{i,k}, X_{i,k+1})\}$;

  end

  $U \leftarrow U \cup \{(X_m, X_{i,1})\}$;

  /* $\forall C_j \in \mathcal{C}, \mathcal{O}$ *induces on* $C_j \cap M$ *a total order* */

end

for C_j *such that* $\{C_i, C_j\} \in E$ **do**

  Depth-first-search(C_j);

end

The order $\mathcal{O}$ built by Algorithm 1 is such that, for any constraint C_i of the hyper-tree, the projection of $\mathcal{O}$ on $Var(C_i)$ defines a total order $\mathcal{O}_i$. We can thus build a linear automaton $\mathcal{A}_i$ for each constraint C_i and then compose these automata around $\mathcal{O}$ following the following principles of asymetric composition:

Asymetric composition Let $\mathcal{A}$ a tree-driven automaton whose support $\mathcal{T}$ is oriented by some partial order $\mathcal{O}$ and $\mathcal{A}_l$ a linear automaton based on some total order $\mathcal{O}_l$ of a subset $Q_l \subseteq Q_{\mathcal{T}}$. When $\mathcal{O}_l$ is induced by $\mathcal{O}$ (i.e. $\text{Proj}(\mathcal{O}, Q_l) = \mathcal{O}_l$), any word recognized by $\mathcal{A}$ can be soundly projected on $\mathcal{T}_l$ and thus tested w.r.t. $\mathcal{A}_l$.

Let us define $\mathcal{A} \bigotimes \mathcal{A}_l$ as the automaton that accept all and only the words accepted by both automata, i.e. such that:
$\mathcal{L}(\mathcal{A} \bigotimes \mathcal{A}_l) = \{w \in \mathcal{L}(\mathcal{A})$ such that $projection(w, \mathcal{T}_l) \in \mathcal{L}(\mathcal{A}_l)\}$. Algorithm 2 shows how to build $\mathcal{A} \bigotimes \mathcal{A}_l$ when $\mathcal{A}_l$ is a linear automaton ($\mathcal{T}_l$ is a subchain of $\mathcal{T}$).

Algorithm 2: Performing the asymetric combination $\mathcal{A} \bigotimes \mathcal{A}_l$

Data: $\mathcal{A}$ a tree driven automata oriented by $\mathcal{O}$, $\mathcal{A}_l$ a linear automata built on $\mathcal{O}_l$, $\mathcal{O}_l$ being induced by $\mathcal{O}$

Result: $\mathcal{A} \bigotimes \mathcal{A}_l$

$I := \mathcal{O}_l$'first; $F := \mathcal{O}_l$'last;

$\mathcal{X}' := Proj(\mathcal{O}, \{i \in Q_{\mathcal{T}}, I \leq_{\mathcal{O}} i \leq_{\mathcal{O}} F\}$

/* *All the levels of the support between the initial and the final levels of the linear automata; may contain extra-levels* */

if $\mathcal{X}' \neq \mathcal{O}_l$ **then**

/* *If $\mathcal{O}$ inserts extra levels between levels I and F, extend the linear automaton* */

$\mathcal{A}'_l := extend(\mathcal{A}_l, \mathcal{X}')$;

end

else

$\mathcal{A}'_l := \mathcal{A}_l$

end

/* *The (linear) restriction of the tree automaton to the variables under concern* */

$\mathcal{A}' = proj(\mathcal{A}, \mathcal{X}')$;

/* *Conjunctive composition of linear automata* */

$\mathcal{A}" = \mathcal{A}'_l \bigwedge \mathcal{A}'$;

/* *Re-insertion in the Tree* */

Let $\mathcal{B}$ be a copy of $\mathcal{A}$.

Remove from $\mathcal{B}$ all the states and transitions that belong to $\mathcal{A}'$

Insert $\mathcal{A}"$ into $\mathcal{B}$: Add to $\mathcal{B}$ all the states and transitions that belong to $\mathcal{A}"$ and replace the edges $(q, q_i) \in E_I$ (resp. $(q_f, q \in E_F)$) by $(q, (q_i, q'))$(resp. $((q_f, q'), q)$)

Now, let $\mathcal{O}$ the total order obtained by Algorithm 1 and $\mathcal{T}$ the corresponding support. For each constraint $C_i \in \mathcal{C} = \{C_1, \ldots, C_m\}$, let us build a linear automaton $\mathcal{A}_i$ on based the projection of $\mathcal{O}$ on the variables of C_i (recall that this defines a total order). Let also A_ϕ be the automaton of support $\mathcal{T}$ accepting all the assignments in $D_1 \times \cdots \times D_n$ — it is defined by as follows:

- For $i = 0, n$, set $Q_i = \{q_i\}$
- for any (i, j) in $E_{\mathcal{T}}$, $E_{i,j} = Q_i \times Q_j$ (on edge per level).
- for any (i, j) in $E_{\mathcal{T}}$, set $\sigma(q_i, q_j) = D_j$(the transition recognizes every value in D_j).

It is obvious that $Sol(\Pi) = \mathcal{L}(A_\phi \bigotimes A_1 \bigotimes \cdots \bigotimes A_m)$. So, the iterative composition of the A_i into an automata initialized to A_ϕ will build an automaton of support $\mathcal{L}$ that recognizes $Sol(\Pi)$. The normalization (and other reductions operations e.g. trimming and fusion) of this automaton yield an exploitable tree-driven automaton.

Notice that, for each linear automaton associated to C_i, its width (maximal number of states on a level) is at most K_i, the number of tuples satisfying the constraint C_i (and $K_i \leq d^{arity(Ci)}$, d being the size of the domains), and its size is at most $\mathcal{O}(arity(C_i) \times K_i)$.

Let W be the maximal width of the linear automaton associated to the constraints. In $\mathcal{O}$, each constraint C_k has an associated path P_k; for each variable X_i, let $nbp(i)$ be the number of such paths which contain X_i (in Figure9 for instance, we have $np(4) = 4$: 4 belongs to three constraints, and the path associated to C_1 contains 4). It can be shown that:

Proposition 6. *When $\Pi = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ is an hyper tree of constraints, the number of states at each level i in automaton based on the support computed by the Algorithm 1 is bounded by $\mathcal{O}(W^{nbp(i)})$.*

Proof. When performing the assymetric combination, the extension step applied to the automaton $\mathcal{A}_l$ does not modify $width(\mathcal{A}_l)$. The projection step (linear restriction of a tree automaton on a path) does no more increase the width. This is also true for the last step (re-insertion in tree automaton). The key step is the composition of two linear automata. In this step two linear automata are composed: $\mathcal{A}_l$, associated to a constraint C_l, and $\mathcal{A}$ ', the projection of $\mathcal{A}$ on the variables of C_l. The states at level i of $\mathcal{A}$ ' are tuples $(x_1, x_2, \ldots, x_p)$ where p is the number of constraints involving X_i taken into account in $\mathcal{A}_l$. In the automaton built by the composition operation, the states at level i are then tuples $(x_1, x_2, \ldots, x_{p+1})$ (see Section 2). So, for each constraint involving X_i, the number of states is at most increased by a factor W.

5.3 General case

When the original CSP is not directly an hyper-tree, we will build a cluster tree covering the graph (see for example [5]) and then compute the linear automata corresponding to each hyper-edge and finally combine then by means of the algorithms described in the previous section. Notice that the computation of the linear automata does not explicitly builds the hyper constraints: for each hyper-edge, one will rather build the CSP involving all its variables and all the constraints included in it. This classical CSP is then compiled in a linear automaton, following the approach of [18] (each constraint is individually compiled, and the resulting automata are then composed using classical algorithms of automata composition).

Since tree clustering is an NP-hard process (recall that finding the hyper-tree that minimize the size of the larger a hyper-edge is itself a NP-hard problem, and that, in the wost case, this edge may involve all the variables), the compilation of general CSPs may fail in practice. However it should be noticed that, in product configuration, the CSP is usually “close” to an hyper-tree, because of the decomposition of the product into configurable sub-components.

The model of "Composite CSPs" [12] well suits the expression of such a products: in this framework, some values of some variables are itself subproblems and recursively. The variables in a subproblem are connected to the variables of the upper problems only. A composite CSP should be easily compiled into a tree driven automata, considering that the composite model implicitly defines an hyper-tree of constraints. The CSP Π_0 at the top of the hierarchy as well as each sub-CSP Π_i of Π_0 defines an hyper-edge (involving: the variables of Π_i and all the variables of the upper CSPs connected to at least one variable of Π_i). The corresponding linear automaton is obtained by the compilation Π_i itself, i.e. of the set of constraints relating variables of Π_i to each other or to variables of the upper CSP. The dynamic characteristic of composite CSP could be handled by the dynamical "mounting" of the sub-automata, upon the selection by the user of a "meta-value". But this is out of the scope of the current paper.

6 Conclusion

Solving decision support problems that are interactive in essence implies achieving efficiently some reasoning tasks like e.g. the maintaining of global consistency; as these tasks are highly combinatorial in the general case, a possible approach is to push the cost into an off-line pre-computation step; [1] propose to compile the CSP under the form of a classical automaton. Under this form, the usual reasoning has a complexity polynomial in the size of the automaton. It suffers of two drawbacks: first, the size of the automaton can be very large, and finding a good order of the variables which minimizes it relies heavily on heuristics; secondly, the linear structure of the automaton does not really suit the decomposition to of products into components and sub-components.

In this paper, we have introduced tree-driven automata as a new compilation target. Tree-automata allow to take better advantage of the structure of the CSP, need less space (in the case of tree-structured CSPs, space is linear in the size of the micro-structure whereas it can be exponential when using a linear target structure), are more suited to the addition and removing of subcomponents, and yet keep polynomial complexity for usual reasoning tasks.

In the framework of automated reasoning, compilation methods have been proposed (see [4], which can be compared with ours, with the slight difference that they are restricted to boolean variables). Tree driven automata do not correspond exactly to any of these, but they take advantage of the same properties: decomposability, "read once" property, order on the variables and determinism. This suggests to study other classes of representation, e.g. generalized tree driven automata (which should relax the partial order to a "read once" property) or multi-valued DNNF.

Appendix: proof of the property 1

We recall that for $k \geq 3, k < d-1$, the CSP $\Pi(h,k,d) = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ is defined by:

- its constraint graph is the complete k-ary tree with height h (see Figure 3 for an example with $k = 3$ and $h = 2$)
  Therefore, we have: $n = |\mathcal{X}| = \sum_{i=0}^{h} k^i = (k^{h+1} - 1)/(k - 1)$.
- $D_1 = \ldots = D_n = D$ with $|D| = d$
- each constraint relation is the relation $\neq$

We have to proof that, for any total order $\mathcal{O}$ on $\mathcal{X}$, the size of $\mathcal{A} = (Q, E)$, the deterministic automaton representing the set of solution of $\Pi(h, k, d)$, is exponential w.r.t. h, k and d.

Consider that $\mathcal{O}$ has been fixed and let $var(i)$ denotes the variable of $\mathcal{X}$ which has number i in $\mathcal{O}$ and let s_q be an assignment of $var(1), \ldots, var(i)$.

Definition 9.

- *The **right-language of** s_q, denoted $\mathbf{L_R(s_q)}$, is defined by:*
  $L_R(q) = \{s' \ / \ s'$ *partial assignment of* $\{var(i + 1), \ldots, n\}$
  which extends the partial assignment s_q *to a solution of* $\Pi\}$.
- ***Equivalent assignments:***
  Two partial assignments s_p *and* s'_p *of the same set* $var(1), \ldots, var(i)$ *are equivalent iff they share the same right language.*

Let p (resp. p') be a path for the root to level i representing s_p (resp. s'_p) in the automaton and q (resp. q') the last state of this path (q and q' are thus states at level i); notice that when the automaton is deterministic, then this path is unique. Anyway, if the automaton is minimal, then $q = q'$ – otherwise, they could be merged thus leading to a smaller automaton.

Therefore, N_i, the number of states at level i deterministic automaton is precisely the number of equivalence classes in $D_1 \times \cdots \times D_i$. The size of the DFA lowered by $\sum_i N_i$ – let us now estimate a lower bound of N_i.

The assignments of $\{var(i+1), \ldots, var(n)\}$ extending an assignment of the i first variables depends only on the subset of these i variables having a neighbor in $\{var(i + 1), \ldots, var(n)\}$. In the context of graph layout problems [6], this set is called *vertex separation at level* i *for the order* $\mathcal{O}$ and denoted $Sep(\mathcal{O}, i)$.

Similarly, given an unassigned variable $var(j)$ ($j > i$), its possible values depend on the values given to the variables of $\{var(1), \ldots, var(i)\}$ adjacent to j in the constraint graph — this set is called *vertex separation at level* i *for* j – *w.r.t.* $\mathcal{O}$. Formally:

Definition 10. *Let* O *be a total order on* $\mathcal{X}$*, and let* C *be the set of edges of the constraint graph. For* i*,* $1 < i < n$*, we define :*

- $Sep(\mathcal{O}, i) = \{\ l \leq i / \exists\ k > i \text{ with } \{var(l), var(k)\} \in C\ \}$
- $Sep_j(\mathcal{O}, i) = \{\ l\ /\ l \leq i \text{ and } \{var(l), var(j)\} \in E\ \}$

Obviously, $Sep_j(\mathcal{O}, i) \subseteq Sep(\mathcal{O}, i)$. $Sep_j(\mathcal{O}, i)$ may be empty if $var(j)$ isn't linked w.r.t. the i first variables.

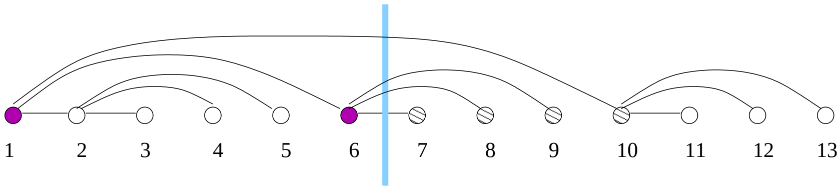

Fig. 11. vertex separation at level 6 for $\Pi(2,3,d)$

When the order $\mathcal{O}$ is implicit, for simplicity sake, we write $Sep(i)$ rather than $Sep(\mathcal{O},i)$. For this example, $Sep(6) = \{1,6\}, Pos(6) = \{7,8,9,10\}$ and $Sep_7(6) = Sep_8(6) = Sep_9(6) = \{6\}, Sep_{10}(6) = \{1\}$.

Let us now study the consequences of semantics of our constraints of difference. Since $Sep_j(i)$ is the set of neighbors of j in $\{var(1),\ldots,var(i)\}$, and the constraints state that the value of $var(j)$ has to be different of the values of its neighbors. Thus given a locally consistent assignment s of $\{var(1),\ldots,var(i)\}$ and $j > i$ (unassigned). it holds that the value of $var(j)$ must be different from those given to the variables in $Sep_j(i)$, i.e. cannot belong to $\{s(k)/k \in Sep_j(i)\}$ – let us denote $UnAuth(s,j)$ this set of unauthorized values. It is actually possible to show that two assignments sharing the same set of unauthorized values are equivalent.

Proposition 7. *Let s_1 and s_2 two assignments of level i. Then,*

$$L_R(s_1) = L_R(s_2) \Leftrightarrow \forall j \in Pos(i), \quad UnAuth(s_1,j) = UnAuth(s_2,j).$$

Proof. $\Leftarrow$ Let s be a solution extending s_1. Let s' be the assignment identical to s_2 on $\{1,\ldots,i\}$ and identical to s on $\{i+1,\ldots,n\}$. Let $\{var(l), var(j)\}$ be any edge such that $l \leq i < j$. By property **??**, $s(j) \notin UnAuth(s_1,j)$ which is equal to $UnAuth(s_2,j)$ by hypothesis; then $s(j) \notin UnAuth(s_2,j)$; we have thus $s(j) \neq s_2(l)$ and, by definition of s', this implies $s'(j) \neq s'(l)$; so the constraint between $var(j)$ and $var(l)$ is satisfied and s' is a solution, $s(\{i+1,\ldots,n\}\ ) \in L_R(s_2)$, and $L_R(s_1) \subseteq L_R(s_2)$. By symmetry, $L_R(s_1) = L_R(s_2)$.

$\Rightarrow$ Suppose that $\exists j > i$ such that $UnAuth(s_1,j) \not\subset UnAuth(s_2,j)$ and let $a \in UnAuth(s_1,j) \setminus UnAuth(s_2,j)$. As any variable of the CSP has at most $k+1$ neighbors, and as $k < d-1$, there exists a solution s extending s_2 such that $s(j) = a$; we have $s(\{i+1,\ldots,n\}\ ) \in L_R(s_2)$. But, since $a \in UnAuth(s_1,j), \exists l \leq i$ s.t. $l \in Sep_j(i)$ and $s(l) = s_1(l) = a$. The constraint between $var(j)$ and $var(l)$ is not satisfied, and the extension of s_1 by $s(\{i+1,\ldots,n\}\ )$ is not a solution. Then $L_R(s_1) \neq L_R(s_2)$.

Consider s_1 and s_2 two locally consistent assignments of i variables. s_1 and s_2 belong to the same class iff for all $\forall j > i$ $UnAuth(s_1,j) = UnAuth(s_2,j)$, i.e.

$\forall j > i$ $\{s_1(k)/k \in Sep_j(i)\} = \{s_2(k)/k \in Sep_j(i)\}$. We can exclude the j not linked to variables in i (for those $UnAuth(s_1, j) = UnAuth(s_2, j) = \emptyset, \forall s_1, s_2$).

When the sets $Sep_j(i)$ are all disjoint, the number of equivalence classes is easy to compute; unfortunately, this is not always true (it is the case for example in Figure 11, as $Sep_7(6) = Sep_8(6)$.

Nevertheless, we can obtain a lower bound by using a subset of $Sep(i)$.

Let us consider B_i, the bipartite subgraph of $(\mathcal{X}, \mathcal{C})$, with vertices $Sep(i) \cup Pos(i)$, and with edges $\{(k, l)$ s.t. $k \in Sep(i), l \in Pos(i), k$ and l adjacent $\}$.

A matching M of B_i is a subset of its edges such that for each vertex x of B_i, at most one edge of M is incident on x.

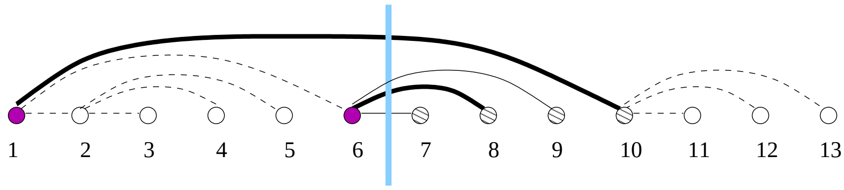

Fig. 12. We use again the example of Figure 11; the continuous edges are the the edges of B_6; the set of thick edges is a matching.

A vertex x is *matched* by matching M if some edge in M is incident on x; otherwise, x is *unmatched*. We denote by M_S (resp. M_P) the vertices of $Sep(i)$ (resp. $Pos(i)$) matched by M.

Proposition 8. *Let M be a matching of B_i, and let s_1 and s_2 be two locally consistent assignments of M_S. Then,*

$s1 \neq s_2 \quad \Rightarrow \quad \exists s_1'$ *and* s_2', *two locally consistent assignments of* $Sep(i)$
extending respectively s_1 *and* s_2 *which are not equivalent.*

Proof.
Let s_1 and s_2 be two locally consistent assignments of M_S such that $s1 \neq s_2$. The, there exists x_j in M_S such that $s_1(x_j) \neq s_2(x_j)$. Let y_j be the neighbor of x_j in M. We have to consider the cases:

a) $UnAuth(s_1, y_j) \neq UnAuth(s_2(y_j)$, we build s_1' (resp. s_2') by assigning to each unmatched neighbor of y_j a value of $UnAuth(s_1, y_j)$ (resp. of $UnAuth(s_2(y_j)$; we have then $UnAuth(s_1', y_j) = UnAuth(s_1, y_j)$ and $UnAuth(s_2', y_j) = UnAuth(s_2, y_j)$, hence $UnAuth(s_1', y_j) \neq UnAuth(s_2', y_j)$; so $s'1$ and $s'2$ are not equivalent.
b) $UnAuth(s_1, y_j) = UnAuth(s_2(y_j)$. In this case, since $s_1(x_j) \neq s_2(x_j)$, x_j cannot be the only neighbor of y_j in B_i.

b-1) either y_j has at least one unmatched neighbor in B_i; as y_j has degree at most $k+1$, it has at most k matched neighbors, so $UnAuth(s_1, y_j)$ has at most k distinct values; since $k < d-1$, we can find two values a, b in $\mathcal{D}$ which does not belong to $UnAuth(s_1, y_j)$; let us extend s_1 to s'_1 by taking $s'_1(u) = a$ for each unmatched neighbor u of y_j in B_i; similarly, we extend s_2 to s'_2 by taking $s'_2(u) = b$; so, $UnAuth(s'_1, y_j) = UnAuth(s_1, y_j) \cup \{a\}$ and $UnAuth(s'_2, y_j) = UnAuth(s_2, y_j) \cup \{b\}$ are different, and $s'1$ and $s'2$ are note equivalent.

b-2) or all the neighbors (at least 2) of y_j are matched; as $s_1(x_j) \neq s_2(x_j)$ and $UnAuth(s_1, y_j) = UnAuth(s_2(y_j))$, there exists x_{j_1} a neighbor of y_j such that $s_1(x_{j_1}) \neq s_2(x_{j_1})$, and let $\{x_{j_1}, y_{j_1}\}$ the associated edge in M. We can then build a path $\{x_j, y_j, x_{j_1}, y_{j_1}, \ldots, x_{j_l}, y_{j_l}\}$ where all the vertices are distinct, and such that:
$\{x_{j_k}, y_{j_k}\} \in M, \forall j_k \leq j_l$ As this sequence is bounded
$s_1(x_{j_k}) \neq s_2(x_{j_k}) \forall j_k < j_l$
$UnAuth(s_1, y_{j_k}) = UnAuth(s_2, y_{j_k}) \forall j_k < j_l$
all the neighbors of y_{j_k} are matched $\forall j_k < j_l$
by the size of M, there exists j_l such that x_{i_l} and y_{i_l} verify *a)* or *b-1)*.

Two consequences follow from this property:

- the number of different assignments of M_S is a lower bound of the number of non equivalent assignments of the first i variables
- moreover, it is easy to see that the number of different locally consistent assignments of M_S is at least $(d-1)^{|M_S|}$.

So, to finish the proof, we need only to check, that, for any order $\mathcal{O}$, there is a level i having a matching of size large enough.

Proposition 9. *Let $(\mathcal{X}, \mathcal{C})$ be a complete k-ary tree with height h, $k \geq 3$. Then, for any order $\mathcal{O}$ on $\mathcal{X}$, there exists c such that B_c has a matching of size greater or equal to h.*

Proof. The proof is by induction on the height of the tree.
The result is trivially true for a tree of height 1.
Let us consider a tree T of height h, with the root r, and its k subtrees $T_1, \ldots, T_k$ with roots $r_1, \ldots, r_k$. Let $\mathcal{O}$ be a total order on the vertices of T. We denote by $\mathcal{O}(x)$ the rank of the vertex x in $\mathcal{O}$, and, for any T_u, its vertices lie in an interval I_u of $\mathcal{O}$:$I_u = [min(I_u), max(I_u)]$ where $min(I_u) = \min_{x \in T_u}(\mathcal{O}(x))$, and $max(I_u) = \max_{x \in T_u}(\mathcal{O}(x))$. $\mathcal{O}$ induces on any T_u an order $\mathcal{O}_u$, and, by the induction hypothesis, there exists c_u such that $B_{c_u}(T_u)$ has a matching M_u of size at least h. Let $\mathcal{O}(c_u)$ be the rank in $\mathcal{O}$ corresponding to the rank c_u in T_u. We have to consider the two following cases:

- either there exists two subtrees T_u and T_v such that $min(I_v) < \mathcal{O}(c_u) < max(I_v)$. As T_v is connected, there exists an edge $\{x, y\}$ of T_v such that $\mathcal{O}(x) < \mathcal{O}(c_u) < \mathcal{O}(y)$; we get a matching in $B_{\mathcal{O}(c_u)}$ of size at least h, by adding this edge to M_u.

– or, for every subtrees T_u and T_v $\mathcal{O}(c_u) \not{i}n I_v$; this implies that , $forall u \in [1,k]$, $\mathcal{O}(c_u)$ and $\mathcal{O}(c_u+1)$ are consecutives in $\mathcal{O}$; let us consider T_u, T_v, T_w such that $\mathcal{O}(c_u) < \mathcal{O}(c_u+1) < \mathcal{O}(c_v) < \mathcal{O}(c_v+1) < \mathcal{O}(c_w) < \mathcal{O}(c_w+1)$. We have then $\mathcal{O}(r_u) < \mathcal{O}(c_v)$ and $\mathcal{O}(r_w) > \mathcal{O}(c_v+1)$. If $\mathcal{O}(r) < \mathcal{O}(c_v)$, we get a matching of size at least h by adding to M_v the edge $\{r, r_w\}$; the case $\mathcal{O}(r) > \mathcal{O}(c_v)$ we get again a matching of size at least h by adding to M_v the edge $\{r, r_v\}$

As $h = O(log(n))$, this is larger than any polynomial in d: We have shown that the CSPs $\Pi(h,d,k)$ with $k \geq 3$ and $k < d-1$ do not have a polynomially-sized representation by a deterministic linear automaton.

References

1. J. Amilhastre, H. Fargier, P. Marquis, Consistency restoration and explanations in dynamic CSPs - Application to configuration. Artificial Intelligence, 135, 199–234, 2002.
2. J. Amilhastre. *Représentation par automate de l'ensemble des solutions de problèmes de satisfaction de contraintes.* PhD thesis, Université du Languedoc, Montpellier, 1 1999.
3. M. Cadoli and F. M. Donini, A survey on knowledge compilation, *AI Communications*, **10**, 137–150, 1997.
4. A. Darwiche, P. Marquis. A perspective on knowledge compilation. In IJCAI 2001,175–182.
5. R. Dechter, J. Pearl, 'Tree clustering for constraint networks', in *Artificial Intelligence*, **38**, 353–366, 1989.
6. J. Diaz, J. Petit, M. Serna, A survey of graph layout problems, *ACM Computing Surveys*, **34(3)**, 313–356, 2002.
7. J. Ellis, I.H. Sudborough, J. Turner, The vertex separation and search number of a graph, *Information and computation*, **113**, 50–79, 1979.
8. E. Gelle and R. Weigel, 'Interactive configuration using constraint satisfaction techniques', in *Artificial Intelligence and Manufacturing Research Planning Workshop, AAAI Technical Report FS-96-03*, 37–44, 1996).
9. J. Hoey, R. St-Aubin, A. Hu, and C. Boutilier. SPUDD: Stochastic planning using decision diagrams. In Proceedings UAI'99, 279–288, 1999.
10. Sanjay Mittal et Brian Falkenhainer. Dynamic constraint satisfaction problems. In *Proceedings of AAAI-90*, pages 25–32, Boston, MA, 1990.
11. B. Pargamin. Extending Cluster Tree Compilation with non-Boolean variables in Product Configuration: a Tractable Approach to Preference-based Configuration, in *Proceedings of the IJCAI'03 Workshop on Configuration*, Acapulco, Mexico, 2003.
12. Daniel Sabin et Eugene C. Freuder. Composite constraint satisfaction. In *Artificial Intelligence and Manufacturing Research Planning Workshop*, pages 153–161, 1996.
13. M. Sabin and E. C. Freuder, 'Detecting and resolving inconsistency in conditional constraint satisfaction problems', in *Proceedings of the AAAI'99 Workshop on configuration*, 90–94, Orlando, Florida, 1999.

14. M. Sabin and E. C. Freuder, 'Detecting and resolving inconsistency in conditional constraint satisfaction problems', in *Proceedings of the AAAI'99 Workshop on configuration*, 90–94, Orlando, Florida, 1999.
15. D. Sabin and R. Weigel, 'Product configuration frameworks - a survey', *IEEE Intelligent Systems and their applications*, 42–49, 1998.
16. T. Soininen and E. Gelle, 'Dynamic constraint satisfaction in configuration', in *Proceedings of the AAAI'99 Workshop on configuration*, 95–100, Orlando, Florida, (1999).
17. M. Stumptner, 'An overview of knowledge-based configuration', *AI Communications*, 111–125, (1997).
18. N. R. Vempaty. Solving constraint satisfaction problems using finite state automata, in *Proceedings of AAAI'92*, 453–458, 1992.
19. R. Weigel and Boi Faltings, 'Compiling constraint satisfaction problems', *Artificial Intelligence*, **115**, 257–287, 1999.