Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Covering Sharing Trees: A Compact Data Structure for Parameterized Verification

Giorgio Delzanno[1], **Jean-François Raskin**[2] ⋆, **Laurent Van Begin**[2] ⋆⋆

[1] Dipartimento di Informatica e Scienze dell'Informazione,
Università di Genova,
via Dodecaneso 35, 16146 Genova, Italy
e-mail: `giorgio@disi.unige.it`

[2] Département d'Informatique,
Université Libre de Bruxelles,
Blvd Du Triomphe, 1050 Bruxelles, Belgium
e-mail: {`jraskin`,`lvbegin`}`@ulb.ac.be`

Abstract.
The *control state reachability problem* is decidable for *well-structured infinite-state systems* like (Lossy) Petri Nets, Vector Addition Systems, and Broadcast Protocols. An abstract algorithm that solves the problem is the *backward reachability algorithm* of [1,21]. The algorithm computes the closure of the *predecessor operator* with respect to a given upward-closed set of target states. When applied to this class of verification problems, symbolic model checkers based on constraints like [7,29] suffer from the *state explosion problem.*

In order to tackle this problem, in [14] we have introduced a new data structure, called *Covering Sharing Trees*, to represent in a compact way collections of infinite set of system configurations. In this paper, we will study the theoretical complexity of the operations over Covering Sharing Trees needed in symbolic model checking. We will also discuss several optimizations that can be used when dealing with Petri Nets. Among them, in [15] we have introduced a *new heuristic rule* based on *structural properties* of Petri Nets that can be used to efficiently prune the search during symbolic backward exploration. The combination of these techniques allowed us to turn the abstract algorithm of [1,21] into a practical method. We have evaluated the method on several finite- and infinite-state examples taken from the literature [2,18,20,30]. In this paper we will compare the results we obtained in our experiments with those obtained using other finite and infinite-state verification tools.

⋆ This author was partially supported by the FRFC grant 2.4530.02.
⋆⋆ This author was supported by a "First Europe" grant EPH3310300R0012 of the Walloon Region.

Keywords Symbolic Model Checking, Infinite-state Systems, Constraints, Efficient Data Structures, Heuristics, and Static Analysis.

1 Introduction

In recent years many efforts have been made to extend the theoretical results and practical methods developed for verification of finite-state systems (see e.g. [6]) to systems with *infinite*-state space (see e.g. [1,5,4,17,21]). This class of systems comprises well-known examples like Vector Addition Systems [31], extensions of Petri Nets [11,35], Integral Relational Automata [9], Broadcast Protocols [16] and Lossy Petri Nets [5]. The *control state reachability* problem is decidable for all previous systems [1,5,17,19,21]. The abstract algorithm proposed in [1,21] computes the closure of the *predecessor* operator with respect to a given *upward-closed set of states.* Upward-closed set of states can be often used to represent violations of *safety properties* like *mutual exclusion* and *coverability for markings of Petri Nets.*

As in the finite-state case, the success of symbolic model checking for this class of problems depends on the data structures used as implicit representation of sets of states. Over numerical domains, upward-closed sets can be represented via a sub-class of integer arithmetic constraints (see e.g. [1,12]). In this setting the state-space generated by the algorithm of [1,21] can be represented as a large (often huge) *disjunction* of arithmetic constraints. In [12] several different constraint-based verification methods working over integers (based on a solver for Presburger arithmetic [7]) and reals (based on polyhedra [29]) have been tested on verification problems of safety properties of Petri Nets. The examples in [12] would be considered of negligible size in finite-state model checking (e.g. one of the Petri Net has only six transitions and ten places). Despite of the size of the examples, all the methods considered in [12] suffer from the following *symbolic state explosion* problem: the length of the arithmetic formulas used to represent the intermediate results explodes after few iterations of backward exploration. Some of the experiments required (when terminating) execution times in the order of days. Based on these observations, it seems natural to look for BDD-like data structures to compactly represent generalization of Boolean formulas useful for the above mentioned verification problems.

In this paper we propose a new symbolic representation for upward-closed sets based on the sharing trees of Zampuniéris and Le Charlier [37]. Sharing trees are acyclic graphs used to represent large sets of tuples. The intuition behind the choice of this data structure is as follows. An upward-closed set U is determined by its finite set of minimal points (tuples of integers). Thus, we can represent the set U by a sharing tree whose paths correspond to its minimal points. As we will discuss later in this paper, in several practical cases it is more convenient to store a set of *generators* for U, by dropping the minimality condition. The denotation associated with the resulting graph is the *infinite set* of integer tuples generated by the tuples represented by the sharing tree. The data structure with the new interpretation is called *Covering Sharing Tree* (CSTs). All operations needed to implement backward reachability can be implemented symbolically on CSTs. As we will discuss in this paper, this technology allowed us to scale up the size of the examples studied via the backward reachability verification method.

In the rest of this section, we will discuss in detail the technical contribution of this paper.

The logic $\mathcal{U}$ and Covering Sharing Trees In this paper we will first introduce the logic $\mathcal{U}$ in which collections of upward-closed sets can be represented as disjunctive formulas. $\mathcal{U}$-formulas are used for verification problems of infinite-state systems as boolean formulas are used for the finite-state case. We will show that CSTs can be used to obtain compact representations of $\mathcal{U}$-formulas. Specifically, we will show that there exists a $\mathcal{U}$-formula that can be represented using a CST whose size is logarithmic in the size of the formula. In this paper, we will show how basic operations on $\mathcal{U}$-formulas (e.g. conjunction and disjunction) can be implemented symbolically on the corresponding sharing trees. CSTs can be viewed as the BDDs for $\mathcal{U}$-formulas. Furthermore, we will investigate the theoretical complexity of all operations on CSTs needed in symbolic model checking.

Simulation-based heuristics In some practical cases, the CSTs may still become very large, e.g., due to the presence of redundant *generators*. As we will prove in the paper, removing all redundancies from a CST cannot be done in polynomial time. For this reason, we will compute approximated solutions via polynomial time algorithms. The algorithms are based on *simulation relations* defined over nodes of CSTs. The same techniques can be used to give sufficient conditions for the subsumption test of CSTs representing $\mathcal{U}$-formulas. The complete test is coNP-complete in the size of the CSTs.

Structural Analysis and Backward Reachability In order to attack *symbolic state explosion* for Petri Nets of large size, we will combine our CST-based technology with heuristics based on the *Structural Theory* of Petri Nets. More precisely, the heuristic rule we will propose in this paper is based on a *fully automatic* static analysis [34], whose results can be used during the backward reachability algorithm to cut the search space. As the other general purpose heuristics presented in the first part of the paper (e.g. simulation relations), our *structural heuristic* works in polynomial time on the graph structure of CSTs. When combined with the use of CSTs, the heuristic rule allow us to scale up the dimension of the case-studies of one order of magnitude.

Experimental Evaluation As a practical contribution, we will present a set of benchmarks we obtained with an optimized implementation of the CST-library, integrated with the structural heuristic. We have applied the resulting model checking algorithm to a large set of examples of parameterized verification problems that can be solved using decision procedures for coverability of Petri Nets (e.g. mutual exclusion for parametric models like the Mesh and Multipoll examples of [2,30], and semi-liveness for the PNCSA protocol of [20]). In addition to these experiments, we have applied our method to verify safety properties of *finite-state* systems, e.g., some of the above mentioned examples for fixed values of the parameters occurring in the initial marking. In this paper we will compare our results with the results obtained with the specialized tool GreatSPN [10] for computing the reachability set of Petri Nets. As foreseen by Bultan in [8], in most of the cases proving a parameterized property turns out to be more efficient than proving its finite-state instances.

Plan of the Paper. In Section 2 we will introduce some preliminary notions we will use throughout the paper. In Section 3 we will define the logic $\mathcal{U}$ and in Section 4 we will define the operations on formulas of the logic $\mathcal{U}$. In Section 5 we will introduce the notion of *sharing tree.* In Section 6 we will introduce the symbolic representation of $\mathcal{U}$-formulas via CSTs. In Section 7 we will define the symbolic operations for CSTs. In Section 8 we will define simulation relations for nodes of CSTs and discuss their application in the operations for $\mathcal{U}$-formulas. In Section 9 we will illustrate how these technologies can be used for solving *parameterized verification problems.* In Section 10 we will define a new heuristics to prune the search based on the Structural Theory of Petri Nets. In Section 11 we will present an experimental evaluation of our techniques. In Section 12 we will discuss related works and future research directions.

This paper extends the preliminary works of the authors [14,15] with complete proofs and new experiments and comparisons with other methods.

2 Preliminaries

In this paper we will work on sets of tuples of integer values. We will use $\mathbb{Z}$ to denote integer numbers, $\mathbb{N}$ to denote non-negative integer numbers, and D^k to denotes sets of tuples with k components taken from D. Furthermore, we will use $\mathbf{t}$ to denote a tuple $\langle t_1, \ldots, t_k \rangle$ in D^k, i.e., such that $t_i \in D$ for $i : 1, \ldots, k$.

The *component-wise ordering* $\preccurlyeq$ over tuples in $\mathbb{Z}^k$ is defined as follows: $\mathbf{t} \preccurlyeq \mathbf{t}'$ if and only if $t_i \leq t'_i$ for $i : 1, \ldots, k$. When neither $\mathbf{t} \preccurlyeq \mathbf{t}'$ nor $\mathbf{t}' \preccurlyeq \mathbf{t}$, we say that $\mathbf{t}$ and $\mathbf{t}'$ are incomparable.

If $\mathbb{Z}$ is extended with special values like $-\infty$ and $+\infty$, then the ordering is naturally extended by taking $-\infty < c < +\infty$ for any $c \in \mathbb{Z}$. The componentwise ordering $\preccurlyeq$ is extended in the natural way.

Given a tuple $\mathbf{t} \in (\mathbb{Z} \cup \{-\infty\})^k$, we define $cone(\mathbf{t})$ as the *upward-closed set* (or *cone*) generated by $\mathbf{t}$, namely,

$$cone(\mathbf{t}) = \{\mathbf{t}' \in \mathbb{Z}^k \mid \mathbf{t} \preccurlyeq \mathbf{t}'\}.$$

We extend this definition to a set S in the natural way. A set S of tuples is *upward-closed* if and only if $S = cone(S)$.

When restricted to *non-negative* integer values, the ordering $\preccurlyeq$ turns out to be a *well-quasi ordering* [22], i.e., for any infinite sequence $\mathbf{t_1 t_2} \ldots$ of tuples in $\mathbb{N}^k$ there always exists two indices $i < j$ such that $\mathbf{t_i} \preccurlyeq \mathbf{t_j}$. A consequence of this property is that *upward-closed sets* of tuples in $\mathbb{N}^k$ are always *generated* by a *finite sets of minimal tuples* (finite basis). Note that this property does not hold for $\mathbb{Z}^k$.

3 The Logic of Upward-Closed Sets

In this section we will introduce the logic $\mathcal{U}$ that we use to define collections of upward-closed sets. The language is defined on top of a subset of linear integer arithmetic constraints having the form $x \geq c$ for some constant c. The complete language is obtained by adding *disjunction*, *conjunction* and a special operation to *translate* a given coordinate from its current value x_i to $x_i + d$ for some displacement $d \in \mathbb{Z}$. The latter kind of formulas will be useful to concisely represent *transition relations* for systems with integer data paths (e.g. operations like increment and decrement for *counter* machines).

Formally, let $V = \{x_1, \ldots, x_k\}$ be a finite set of variables. The set of $\mathcal{U}$-formulas is defined by the following grammar.

$$\Phi \quad ::= \quad x_i \geq c \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \Phi[x_i + d/x_i],$$

where $c \in \mathbb{Z} \cup \{-\infty, +\infty\}$[1], $d \in \mathbb{Z}$. $\mathcal{U}$-formulas are interpreted over $\mathbb{Z}^k$.

In the following we will use the tuple $\mathbf{t} = \langle t_1, \ldots, t_k \rangle \in \mathbb{Z}^k$ to denote the valuation of the variables $\langle x_1, \ldots, x_k \rangle$ in which x_i is mapped to t_i for $i : 1, \ldots, k$.

We are now ready to introduce some definitions we will use throughout the entire paper.

Definition 1 (Satisfaction Relation). Given a $\mathcal{U}$-formulas Φ, and a valuation $\mathbf{t}$, we define the *satisfaction relation* $\models$ as follows:

- let $\Phi = x_i \geq c$, then $\mathbf{t} \models \Phi$ if and only if $t_i \geq c$;
- let $\Phi = \Phi_1 \wedge \Phi_2$, then $\mathbf{t} \models \Phi$ if and only if $\mathbf{t} \models \Phi_1$ and $\mathbf{t} \models \Phi_2$;
- let $\Phi = \Phi_1 \vee \Phi_2$, then $\mathbf{t} \models \Phi$ if and only if $\mathbf{t} \models \Phi_1$ or $\mathbf{t} \models \Phi_2$;
- let $\Phi = \Psi[x_i + c/x_i]$, then $\mathbf{t} \models \Phi$ if and only if $\mathbf{t}' \models \Psi$ and $\mathbf{t}'$ is such that $t'_i = t_i + c$, whereas $t'_j = t_j$ for any $j \neq i$.

An *extended valuation* is a tuple $\mathbf{t} = \langle t_1, \ldots, t_k \rangle$ with $t_i \in \mathbb{Z} \cup \{-\infty, +\infty\}$. The satisfaction relation is naturally extended to extended valuation.

Definition 2 (Denotation of $\mathcal{U}$-formulas). Given a $\mathcal{U}$-formula Φ, its *denotation* $[\![\Phi]\!]$ is defined as the set of all valuations $\mathbf{t}$ such that $\mathbf{t} \models \Phi$.

In the rest of the paper we will use $\mathcal{U}_k$ to denote the set of $\mathcal{U}$-formulas defined over k variables.

Definition 3 (Subsumption Relation). Given two $\mathcal{U}_k$-formulas Φ_1 and Φ_2, Φ_1 is *subsumed* by Φ_2, written $\Phi_1 \models \Phi_2$, whenever $[\![\Phi_1]\!] \subseteq [\![\Phi_2]\!]$.

We say that two $\mathcal{U}_k$-formulas Φ_1 and Φ_2 are equivalent, written $\Phi_1 \equiv \Phi_2$, if their denotations coincide. As an example, let $k = 3$ and consider the formula $\Phi = ((x_1 \geq 3 \vee x_2 \geq 6) \wedge x_3 \geq 3)[x_1 + 2/x_1]$. The denotation $[\![\Phi]\!]$ contains the tuples $\langle 1, 0, 4 \rangle$ and $\langle -4, 7, 4 \rangle$, whereas it does not contain the tuple $\langle 0, 1, 4 \rangle$.

Definition 4 (Disjunctive form). A $\mathcal{U}_k$-formula in *disjunctive form* has the following form:

$$\bigvee_{i:1,\ldots,n} (x_1 \geq c_{i1} \wedge \ldots \wedge x_k \geq c_{ik}).$$

Any $\mathcal{U}$-formula can be always reduced to an *equivalent disjunctive formula*. Note, in fact, that we can always add formulas like $x_i \geq -\infty$ ($x_i \geq 0$ over non-negative integers) when x_i does not occur in the original formula. Furthermore, we can always push substitutions inwards and eliminate them when applied to atomic constraints as follows:

$$(\Phi_1 \bullet \Phi_2)[x_i + c/x_i] \equiv \Phi_1[x_i + c/x_i] \bullet \Phi_2[x_i + c/x_i]$$
$$(x_i \geq d)[x_i + c/x_i] \equiv x_i \geq d - c,$$
$$(x_i \geq d)[x_j + c/x_j] \equiv x_i \geq d \text{ for } i \neq j.$$

[1] $+\infty$ will be useful to define the empty set a.o. in least fixed point computations.

where $\bullet \in \{\vee, \wedge\}$. As an example, consider again the formula Φ defined as

$$((x_1 \geq 3 \vee x_2 \geq 6) \wedge x_3 \geq 3)[x_1 + 2/x_1]$$

It can be equivalently written as $\Phi_1 \vee \Phi_2$, where

$$\Phi_1 \ = \ x_1 \geq 1 \ \wedge \ x_2 \geq -\infty \ \wedge \ x_3 \geq 3$$

and

$$\Phi_2 \ = \ x_1 \geq -\infty \ \wedge \ x_2 \geq 6 \ \wedge \ x_3 \geq 3$$

By construction, the denotation of a $\mathcal{U}$ formula φ is always an upward-closed set of states, i.e., $cone([\![\varphi]\!]) = [\![\varphi]\!]$. When working over non-negative integers, the following property also holds.

Proposition 1 (Completeness for $\mathbb{N}^k$). *Given $I \subseteq \mathbb{N}^k$, if I is upward-closed, i.e., $cone(I) = I$, then there exists a $\mathcal{U}_k$-formula φ such that $[\![\varphi]\!] = I$.*

Proof. As I is an upward-closed subset of $\mathbb{N}^k$, it has a finite number of minimal elements $\{\mathbf{t}_1, \ldots, \mathbf{t}_n\}$. Thus, we can associate with every tuple $\mathbf{t}_i = \langle t_{i1}, \ldots, t_{ik}\rangle$ the formula $\varphi_i \equiv x_1 \geq t_{i1} \wedge \ldots \wedge x_k \geq t_{ik}$. Clearly, we have that $cone(\mathbf{t}_i) = [\![\varphi_i]\!]$. Thus, the formula φ_I defined as $\varphi_I \equiv \varphi_1 \vee \ldots \vee \varphi_n$ is such that $I = [\![\varphi_I]\!]$. □

To ease the notation, in the sequel of the paper we will use Φ, Ψ, etc. to denote *arbitrary* $\mathcal{U}$-formulas, and φ, ψ, etc. to denote formulas in disjunctive form. We now define the notion of *generator* of a disjunctive formula.

Definition 5 (Generators). The set of *generators* of a *disjunctive* $\mathcal{U}_k$-formula φ, called $gen(\varphi)$, is the set of tuples $\langle c_1, \ldots, c_k\rangle$ with $c_i \neq +\infty$ for any i such that $x_1 \geq c_1 \wedge \ldots \wedge x_k \geq c_k$ is one of the disjuncts in φ.

We define the minimal elements as follows.

Definition 6 (Minimal elements). The set of *minimal elements* $min(\varphi)$ of a *disjunctive* formula φ consists of the minimal elements of $gen(\varphi)$ computed with respect to the component-wise ordering $\preccurlyeq$.

Finally, we say that a disjunctive formula is in *normal form* whenever $gen(\varphi) = min(\varphi)$. The generators in $gen(\varphi) \setminus min(\varphi)$ are called *redundant*.

As an example, consider the formula $\varphi = (x \geq 1 \wedge y \geq 2) \vee (x \geq 3 \wedge y \geq 1) \vee (x \geq 2 \wedge y \geq 0)$. Then, $gen(\varphi) = \{\langle 1,2\rangle, \langle 3,1\rangle, \langle 2,0\rangle\}$, and $min(\varphi) = \{\langle 1,2\rangle, \langle 2,0\rangle\}$, i.e., φ is *not* in normal form. Note that, given a disjunctive formula φ, $[\![\varphi]\!]$ coincides with the union of the denotations $cone(\mathbf{t})$ computed over all $\mathbf{t} \in min(\varphi)$, i.e.,

$$[\![\varphi]\!] = \bigcup_{\mathbf{t} \in min(\varphi)} cone(\mathbf{t}). \tag{1}$$

For a general formula Ψ we define $min(\Psi)$ as the minimal set of tuples with components in $\mathbb{Z} \cup \{-\infty\}$ satisfying

$$[\![\Psi]\!] = \bigcup_{\mathbf{t} \in min(\Psi)} cone(\mathbf{t}). \tag{2}$$

4 Operations on formulas in disjunctive form

4.1 Disjunction

Clearly, formulas in disjunctive form are closed under disjunction. Furthermore, given two formulas in normal form φ_1 and φ_2 we can build a formula ψ in normal form such that $[\![\psi]\!] = [\![\varphi_1]\!] \cup [\![\varphi_2]\!]$ in time $\mathcal{O}(|\varphi_1| \times |\varphi_2|)$ (the time needed to remove redundancies (see section 4.7)).

4.2 Conjunction

Given the $\mathcal{U}_k$-formulas in disjunctive form φ and ψ, the disjunctive formula for $\varphi \ \wedge \ \psi$ is defined as follows:

$$\bigvee_{\mathbf{t} \in gen(\varphi), \mathbf{t}' \in gen(\psi)} \ \bigwedge_{i=1}^{k} x_i \geq max(t_i, t'_i),$$

i.e., $gen(\varphi \wedge \psi) = \{\mathbf{s} \mid \exists \mathbf{t} \in gen(\varphi), \mathbf{t}' \in gen(\psi)$ and $s_i = max(t_i, t'_i),\ i : 1, \ldots, k\}$. The formula $\varphi \wedge \psi$ can be built in time $\mathcal{O}(|\varphi_1| \times |\varphi_2|)$. In general the result of this operation might not be in normal form.

4.3 Substitution

The formula $\varphi[x_i + c/x_i]$ is equivalent to the formula φ' obtained from φ by replacing every atom $x_i \geq d$ with $x_i \geq d - c$ ($-\infty - c = -\infty$ and $+\infty - c = +\infty$ for any $c \in \mathbb{Z}$), i.e., $gen(\varphi[x_i + c/x_i]) = \{\mathbf{t}' \mid t'_i + c = t_i,\ t'_j = t_j\ j \neq i,\ and\ \mathbf{t} \in gen(\varphi)\}$. The formula $\varphi[x_i + c/x_i]$ can be built in time $\mathcal{O}(|\varphi|)$.

4.4 Satisfiability

Given a valuation $\mathbf{t}$, we first note that $\mathbf{t} \models \varphi$ holds if and only if there exists $\mathbf{t}' \in gen(\varphi)$ such that $\mathbf{t}' \preccurlyeq \mathbf{t}$. Thus, checking $\mathbf{t} \models \varphi$ can be done in time $\mathcal{O}(|\varphi|)$. Checking if a valuation $\mathbf{t}$ satisfies an arbitrary $\mathcal{U}$-formula Φ can be also done in time $\mathcal{O}(|\Phi|)$. In fact, we simply have to apply the definition of $\models$ by inspecting on the structure of Φ.

4.5 Subsumption

Let φ and ψ be two $\mathcal{U}_k$-formulas in disjunctive form. We can check $\varphi \models \psi$ in time $\mathcal{O}(|\varphi| \times |\psi|)$. In fact, $\varphi \models \psi$ holds if and only if for all $\mathbf{t} \in gen(\varphi)$ there exists $\mathbf{t}' \in gen(\psi)$ such that $\mathbf{t}' \preccurlyeq \mathbf{t}$. As stated in the following theorem, the subsumption test is much harder for generic $\mathcal{U}$-formulas.

Theorem 1. *Given two $\mathcal{U}_k$-formulas Φ and Ψ, checking that $\Phi \models \Psi$ is* coNP*-hard in the size of the formulas.*

Proof. We use a reduction from the *validity* problem (see e.g. [27]) for propositional formulas in disjunctive normal form:

> *Given a propositional formula $F = D_1 \vee \ldots \vee D_n$ where each disjunct D_i is a conjunction of literals over the set of propositional variables $p_1, \ldots, p_m$, is F true for all valuations?*

This problem is coNP-complete [27].
The reduction works as follows. Given a propositional formula F, we build a $\mathcal{U}$-formula $\Phi_{val(F)}$ (polynomial in the size of F) that represents all the valuations that make F true. Then, we reduce validity for F to the subsumption problem $\Phi_{val(true)} \models \Phi_{val(F)}$. Without loss of generality, we assume that *all variables* occur in each disjunct of F either as p, $\neg p$ or $any(p)$. The latter means that p can take any value (note: it is equivalent to say that p *does not* occur in the disjunct). The formula $true$ is equivalent to $any(p_1) \wedge \ldots \wedge any(p_m)$. To encode the valuation for p_i, we use two variables from V, namely v_i and w_i. We use the formula $\Phi_{val(p_i)} = (v_i \geq 1 \wedge w_i \geq 0)$ when p_i evaluates to $true$; $\Phi_{val(\neg p_i)} = (v_i \geq 0 \wedge w_i \geq 1)$ when p_i evaluates to $false$. Note that neither $\Phi_{val(p_i)} \models \Phi_{val(\neg p_i)}$ nor $\Phi_{val(\neg p_i)} \models \Phi_{val(p_i)}$. We use the disjunction $\Phi_{val(any(p_i))} = (v_i \geq 1 \wedge w_i \geq 0) \vee (v_i \geq 0 \wedge w_i \geq 1)$ when p_i can be evaluated either to $true$ or $false$. The previous encoding allows us to keep the size of $\Phi_{val(F)}$ polynomial in the size of F.

Formally, the formula $\Phi_{val(F)}$ is defined by induction on F as follows:

- $\Phi_{val(D_1 \vee D_2)} = \Phi_{val(D_1)} \vee \Phi_{val(D_2)}$;
- $\Phi_{val(C_1 \wedge C_2)} = \Phi_{val(C_1)} \wedge \Phi_{val(C_2)}$;
- $\Phi_{val(p_i)} = (v_i \geq 1 \wedge w_i \geq 0)$;
- $\Phi_{val(\neg p_i)} = (v_i \geq 0 \wedge w_i \geq 1)$;
- $\Phi_{val(any(p_i))} = (v_1 \geq 0 \wedge w_1 \geq 1) \vee (v_1 \geq 1 \wedge w_1 \geq 0)$.

Based on this construction, we obtain that

> F *is valid if and only if* $\Phi_{val(true)} \models \Phi_{val(F)}$.

Since the reduction is *polynomial*, we obtain that the subsumption problem for $\mathcal{U}$-formulas is coNP-hard. □

To prove coNP easiness, we first need the following lemma.

Lemma 1. *Let Φ be an $\mathcal{U}_k$-formula without substitution and let $const(\Phi)$ be the set of integral constants in Φ. Then, for all $\mathbf{t} \in min(\Phi)$, we have $t_i \in const(\Phi) \cup \{-\infty\}$ for any $1 \leq i \leq k$.*

Proof. We prove the proposition by contraposition. Let $\mathbf{t} \in \mathbb{Z}^k$ be a valuation with $t_i \notin const(\Phi)$ for some i. We will show that $\mathbf{t} \notin min(\Phi)$. We first isolate constraints on x_i imposed by the components t_j of $\mathbf{t}$ for $j \neg i$. This is done by constructing the formula Φ' as follows. Replace every constraint of the form $x_j \geq c$ by $true$ if $t_j \geq c$, $false$ otherwise for any $j \neq i$. After simplification, Φ' can be of three forms.

1. $\Phi' \equiv false$. In that case $\mathbf{t} \not\models \Phi$ and we conclude that $\mathbf{t} \notin min(\Phi)$.
2. $\Phi' \equiv true$. In that case, consider $\mathbf{t}'$ constructed from $\mathbf{t}$ by replacing t_i by $-\infty$, $\mathbf{t}'$ is such that $\mathbf{t}' \models \Phi$ and as $\mathbf{t}' \prec \mathbf{t}$ we conclude that $\mathbf{t} \notin min(\Phi)$.
3. Φ' is a boolean combination of constraints of the form $x_i \geq c$. In that case, we can further simplify Φ' to obtain an equivalent formula Φ'' as follows. In a bottom-up manner, we replace $x_i \geq c_1 \wedge x_i \geq c_2$ by $x_i \geq max(c_1, c_2)$ and $x_i \geq c_1 \vee x_i \geq c_2$ by $x_i \geq min(c_1, c_2)$. So, we finally obtain $\Phi'' \equiv x_i \geq d$, and we know that $d \in const(\Phi)$ by construction. Then there are two cases. Either $d > t_i$ and $\mathbf{t} \not\models \Phi$ and we conclude that $\mathbf{t} \notin min(\Phi)$ or $d < t_i$ and we can construct $\mathbf{t}'$ from $\mathbf{t}$ by replacing t_i by d. We have $\mathbf{t}' \models \Phi$ and as $\mathbf{t}' \prec \mathbf{t}$ we conclude that $\mathbf{t} \notin min(\Phi)$.

Now, we are ready to prove the following result.

Theorem 2. *Given two $\mathcal{U}_k$-formulas Φ and Ψ, checking that $\Phi \models \Psi$ is coNP-easy in the size of the formulas.*

Proof. We prove that the complement of the subsumption problem is in NP. Let us first observe that we can eliminate (in polynomial time) all occurrences of substitutions from a formula. Without loss of generality, let Φ and Ψ be two formulas without occurrences of the substitution operator.

As $[\![\Phi]\!]$ and $[\![\Psi]\!]$ are upward-closed, we have that

$$\Phi \not\models \Psi \text{ iff } \exists \mathbf{t} \in min(\Phi) : \mathbf{t} \models \Phi \text{ and } \mathbf{t} \not\models \Psi$$

Following Lemma 1, we can decide that $\Phi \not\models \Psi$ by guessing a tuple $\mathbf{t}$ in the set of tuples $S_\Phi = \{\mathbf{t} | \mathbf{t} = \langle t_1, \ldots, t_k \rangle, t_i \in const(\Phi) \cup \{-\infty\}, 1 \leq i \leq k\}$ whose size is bounded by $(|\Phi| + 1)^k$ and check in deterministic polynomial time that $\mathbf{t} \models \Phi$ and $\mathbf{t} \not\models \Psi$. □

As a consequence of the previous results, we have the following corollary.

Corollary 1. *Given two $\mathcal{U}_k$-formulas Φ and Ψ, checking that $\Phi \models \Psi$ is coNP-complete in the size of the formulas.*

4.6 Minimization

Given two $\mathcal{U}_k$-formulas in disjunctive form $\varphi = \varphi_1 \vee ... \vee \varphi_n$ and $\psi = \psi_1 \vee ... \vee \psi_m$, the reduction in normal form of φ according to ψ, written $\varphi \div \psi$, is the formula obtained by removing from φ every disjunct φ_i for which there exists a disjunct ψ_j in ψ such that $\varphi_i \models \psi_j$.

Intuitively, if φ and ψ represent part of a collection of states representations (e.g. the set of visited states and the newly computed frontier during symbolic state exploration), then the operation $\div$ allows us to remove all redundant generators from φ with respect to the generators in ψ.

The formula $\varphi \div \psi$ can be constructed in time $\mathcal{O}(|\varphi| \times |\psi|)$. The denotation of $\varphi \div \psi$ is defined as

$$[\![\varphi \div \psi]\!] = \{\mathbf{c} \in cone(\mathbf{t}) \mid \mathbf{t} \in [\![\varphi]\!],\ \mathbf{t} \notin [\![\psi]\!]\}$$

The $\div$ operation can be seen as a kind of difference : only subsets of valuations generated by a disjunct of φ that is not subsumed by ψ are kept.

4.7 Reduction in normal form.

Given a disjunctive formula $\varphi = \varphi_1 \vee \ldots \vee \varphi_n$ we can reduce it in normal form by eliminating all formulas φ_i such that the corresponding generator is not a minimal element of the set defined by φ. The reduction can be done in time $\mathcal{O}(|\varphi|^2)$.

In the following sections we will introduce special data structures, called *sharing trees* [37], and we will show how to apply them to handle large sets of *generators* of formulas in disjunctive form.

5 Sharing Trees

In this paper we specialize the original definition of [37] as follows.

Definition 7 (k-Sharing Tree). A *k-sharing tree* is a rooted directed acyclic graph $(N, val, succ)$, where

- $N = \{root\} \cup N_1 \ldots \cup N_k \cup \{end\}$ is the finite set of *nodes*; N_i is the set of nodes of *layer i* and, by convention, $N_0 = \{root\}$ and $N_{k+1} = \{end\}$;
- $val : N \rightsquigarrow \mathbb{Z} \cup \{-\infty, \top, \bot\}$ is a labeling function for the nodes;
- $succ : N \rightsquigarrow 2^N$ defines the successors of a node.
- Furthermore,
  1. $val(n) = \top$ if and only if $n = root$;
  2. $val(n) = \bot$ if and only if $n = end$;
  3. $succ(end)=\emptyset$;
  4. for $i : 0, \ldots, k$, for all $n \in N_i$, $succ(n) \subseteq N_{i+1}$ and $succ(n) \neq \emptyset$;
  5. for all $n \in N$, for all $n_1, n_2 \in succ(n)$, if $n_1 \neq n_2$ then $val(n_1) \neq val(n_2)$.
  6. for $i : 0, \ldots, k$, for all $n_1, n_2 \in N_i$ s.t. $n_1 \neq n_2$, if $val(n_1) = val(n_2)$ then $succ(n_1) \neq succ(n_2)$.

In other words, a k-sharing tree is an acyclic graph with root and terminal node such that:

- all nodes of layer i have successors in the layer $i+1$ (cond. 4);
- a node cannot have two successors with the same label (cond. 5);
- two nodes with the same label in the same layer do not have the same set of successors (cond. 6).

Definition 8 (Pre-sharing tree). $\mathbf{S}$ is a *pre-sharing tree* whenever it satisfies at least the conditions (1)-(4).

In the rest of the paper we use $root^{\mathbf{S}}$, $N^{\mathbf{S}}$, and $succ^{\mathbf{S}}$ to refer to the *root*, *set of nodes*, *successor relation* of the sharing-tree $\mathbf{S}$, respectively.

Definition 9 (Path). A *path* of a k-sharing tree is a sequence of nodes $\langle root, n_1, \ldots, n_k, end\rangle$ such that $n_{i+1} \in succ(n_i)$ for $i : 1, \ldots, k-1$, $n_1 \in succ(root)$ and, finally, $end \in succ(n_k)$.

Paths represent tuples of size k of integer numbers as specified in the following definition.

Definition 10 (Flat Denotation). The *flat denotation* of a k-sharing tree $\mathbf{S}$, written $elem(\mathbf{S})$, is the set of elements represented by the paths in $\mathbf{S}$. Formally, the set $elem(\mathbf{S})$ consists of all tuples $\langle val(n_1), \ldots, val(n_k)\rangle$ such that $\langle root, n_1, \ldots, n_k, end\rangle$ is a path in $\mathbf{S}$.

Condition 5 and 6 ensure the *maximal sharing* of *prefixes* and *suffixes* among the paths (elements) of a k-sharing tree. The *empty sharing tree* is the sharing tree with only root and end nodes.

Definition 11 (Size of a sharing tree). The *size* of a sharing tree $\mathbf{S}$, noted $|\mathbf{S}|$, is defined as the number of *nodes* plus the number of *edges*.

As shown in [36], thanks to sharing, the cardinality of $elem(\mathbf{S})$ can be exponentially larger than the size of $\mathbf{S}$.

Given a node n of the i-th layer of a k-sharing tree $\mathbf{S}$, the sub-(sharing)tree $\mathbf{S}_n$ rooted at n is the $k-i+1$-sharing tree obtained as follows. We first isolate the graph rooted at n that consists of all the nodes reachable from n (this subgraph has $k-i+1$ layers and a terminal node). Then, we add a layer with the single root node and we set $succ(root) = \{n\}$. From the previous definition, $elem(\mathbf{S}_n)$ consists of all tuples $\langle m, m_{i+1}, \ldots, m_k\rangle$ obtained from the subpath $\langle n, n_{i+1}, \ldots, n_k, end\rangle$ of $\mathbf{S}$ such that $m = val(n)$ and $m_j = val(n_j)$ $j : i+1, \ldots, k$.

As shown in [37], given a set of tuples F of size k, there exists a unique (modulo isomorphisms of graphs) sharing tree such that $elem(\mathbf{S}) = F$. In the same paper the authors defined algorithms for the basic operations on the flat denotations of sharing trees. The table of Figure 1 summarizes the complexity (in the size of sharing trees) of the set operations implemented via sharing trees. In the rest of the paper we will often refer to operations like union, intersection, emptiness, containment, and equality test. In [37], the authors showed that the operations in Figure 1 can be also applied to pre-sharing trees that satisfy at least condition 5.

6 Covering Sharing Trees

As explained in the previous section, sharing trees can be used to compactly represent *finite sets* of tuples. Our goal, however, is to symbolically represent *upward-closed sets* of integer tuples. For this purpose, we need to *lift* the denotation of a sharing tree to a *rich one* in which the tuples stored in its paths represent *generators* for $\mathcal{U}$-formulas in disjunctive form. In the next section we will study the impact of the lifting from flat to rich denotations on the complexity of the symbolic operations. We first show, however, how to use CSTs to represent $\mathcal{U}$-formulas in disjunctive form.

Operation	Effect	Complexity	Algorithm
emptiness test	$\mathsf{is\_empty}(\mathbf{S})$ *if and only if* $elem(\mathbf{S}) = \emptyset$	constant	constant
membership test	$\mathsf{member}(\mathbf{t}, \mathbf{S})$ *if and only if* $\mathbf{t} \in elem(\mathbf{S})$	poly	linear
containment test	$\mathsf{contained}(\mathbf{S}, \mathbf{T})$ *if and only if* $elem(\mathbf{S}) \subseteq elem(\mathbf{T})$	poly	poly
union	$elem(\mathsf{union}(\mathbf{S}, \mathbf{T})) = elem(\mathbf{S}) \cup elem(\mathbf{T})$	poly	poly
intersection	$elem(\mathsf{intersection}(\mathbf{S}, \mathbf{T})) = elem(\mathbf{S}) \cap elem(\mathbf{T})$	poly	poly
minus	$elem(\mathsf{minus}(\mathbf{S}, \mathbf{T})) = elem(\mathbf{S}) \setminus elem(\mathbf{T})$	poly	poly

Fig. 1. Complexity of the operations on sharing trees.

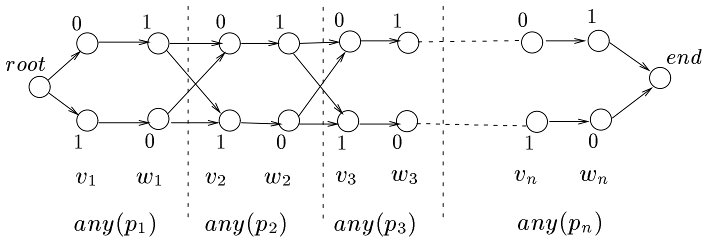

Fig. 2. Sharing tree for a formula in disjunctive form.

Definition 12 (Covering Sharing Tree (CST)). Let φ be a $\mathcal{U}_k$-formula in disjunctive form. The *Covering Sharing Tree* $\mathbf{S}_\varphi$ that represents φ is a k-sharing tree such that $elem(\mathbf{S}_\varphi) = gen(\varphi)$.

Definition 13 (Rich Denotation). The *rich denotation* of $\mathbf{S}_\varphi$ is defined as follows

$$cones(\mathbf{S}_\varphi) = \{\mathbf{t}' \in \mathbb{Z}^k \mid \mathbf{t} \preccurlyeq \mathbf{t}',\ \mathbf{t} \in elem(\mathbf{S}_\varphi)\}.$$

By definition, we obtain that $[\![\varphi]\!] = cones(\mathbf{S}_\varphi)$. The notions of pre-sharing tree and sub-sharing tree naturally extend to CSTs.

Definition 14 (Irredundant CST). The CST $\mathbf{S}_\varphi$ is *irredundant* if and only if there exists no $\mathbf{t} \in elem(\mathbf{S}_\varphi)$ such that $\mathbf{t}' \preccurlyeq \mathbf{t}$ for some $\mathbf{t}' \in elem(\mathbf{S}_\varphi)$ distinct from $\mathbf{t}$.

The following proposition explains the advantages of using sharing trees for representing $\mathcal{U}$-formulas.

Proposition 2. *There exists a formula in disjunctive normal form φ such that the corresponding CST $\mathbf{S}_\varphi$ has a size logarithmic in the size of φ.*

Proof. To prove our claim, we first need some definitions. We define $\mathbf{1}$ as the vector $\langle 1, \ldots, 1\rangle \in \mathbb{N}^{2m}$. Furthermore, given the vectors $\mathbf{t}, \mathbf{t}' \in \mathbb{N}^m$, let $\mathbf{t} \oplus \mathbf{t}'$ be the vector $\langle t_1 + t'_1, \ldots, t_m + t'_m\rangle$.

Consider now the $\mathcal{U}$-formula

$$\Phi = \bigwedge_{i=1}^{m} (v_i \geq 1 \wedge w_i \geq 0) \vee (v_i \geq 0 \wedge w_i \geq 1).$$

The corresponding disjunctive formulas φ (obtained by distributing $\wedge$ over $\vee$) is

$$\bigvee_{\{\mathbf{c},\mathbf{d} \mid \mathbf{c} \oplus \mathbf{d} = \mathbf{1}\}} \bigwedge_{i=1}^{m} (v_i \geq c_i \wedge w_i \geq d_i)$$

This formula has $\mathcal{O}(2^m)$ disjuncts. Also, note that φ is in normal form. In contrast, as shown in Figure 2, the CST $\mathbf{S}_\varphi$ has size logarithmic in φ and polynomial in Φ. In fact, each layer has two nodes and at most four arcs. $\square$

As for sharing trees that are a *canonical form* for finite sets of tuples, irredundant CSTs are a canonical form for upward closed sets of tuples definable by $\mathcal{U}$-formulas as stated in the following theorem.

Theorem 3. *Let Φ be a $\mathcal{U}$-formula and let $\mathbf{S}$ and $\mathbf{S}'$ be two irredundant CSTs. Then, we have $cones(\mathbf{S}) = [\![\Phi]\!]$ and $cones(\mathbf{S}') = [\![\Phi]\!]$ if and only if the CSTs $\mathbf{S}$ and $\mathbf{S}'$ are isomorphic.*

Proof. As $\mathbf{S}$ is irredundant and $cones(\mathbf{S}) = [\![\Phi]\!]$, we know that $elem(\mathbf{S}) = min(\Phi)$, the same holds for $\mathbf{S}'$. So, $\mathbf{S}$ and $\mathbf{S}'$ encode the same set of elements. It is proved in [36] that two such sharing trees are isomorphic. $\square$

In the sequel of the paper we will assume that all the CSTs have k layers.

7 CSTs-based Operations for $\mathcal{U}$-formulas

In this section we show how operations on the disjunctive formulas φ and ψ can be defined symbolically on their representations given by the CSTs $\mathbf{S}_\varphi$ and $\mathbf{S}_\psi$. We use the term *symbolically* because the algorithms that we propose work directly on the graphs $\mathbf{S}_\varphi$ and $\mathbf{S}_\psi$ and not by *enumerating* the tuples that they represent. In the sequel we will show that the complexity of the operations strongly depends on the form of the CSTs we consider. Specifically, we will distinguish between operations that require all arguments to be *irredundant* (abbreviated as

Operation	Abstract property of the operation	Argument type (Irredundant/Generic)	Complexity: Problem	Complexity: Procedure
Emptiness	$\mathsf{CST\_Empty}(\mathbf{S}) = true$ iff $cones(\mathbf{S}) = \emptyset$	GEN	constant	constant
Satisfiability	$\mathsf{CST\_Sat}(\mathbf{t}, \mathbf{S}) = true$ iff $cone(\mathbf{t}) \subseteq cones(\mathbf{S})$	GEN	linear	linear
Substitution	$\mathsf{CST\_Substitute}(\mathbf{S}, c, i) = \mathbf{T}$ s.t. $cones(\mathbf{T}) = cones(\mathbf{S})[t_i + c/t_i]$	GEN	linear	linear
Subsumption	$\mathsf{CST\_Subsumption}(\mathbf{S}, \mathbf{T}) = true$ iff $cones(\mathbf{T}) \subseteq cones(\mathbf{S})$	IRR $\subseteq$ IRR	coNP	exponential
		GEN $\subseteq$ GEN	coNP	exponential
Equivalence	$\mathsf{CST\_Equiv}(\mathbf{S}, \mathbf{T})$ iff $cones(\mathbf{S}) = cones(\mathbf{T})$	IRR $\equiv$ IRR	linear	linear
		GEN $\equiv$ GEN	coNP	exponential
Normal form	$\mathsf{CST\_NormalForm}(\mathbf{S}) = \mathbf{T}$ s.t. $elem(\mathbf{T}) = \{\mathbf{m} \in elem(\mathbf{S}) \mid \nexists \mathbf{m}' \in elem(\mathbf{S}) : \mathbf{m}' \prec \mathbf{m}\}$	GEN	coNP	exponential
Disjunction	$\mathsf{CST\_Union}(\mathbf{S}, \mathbf{T}) = U$ s.t. $cones(U) = cones(\mathbf{S}) \cup cones(\mathbf{T})$	GEN $\cup$ GEN = GEN	polynomial	polynomial
		IRR $\cup$ IRR = IRR	coNP	exponential
Conjunction	$\mathsf{CST\_Intersection}(\mathbf{S}, \mathbf{T}) = \mathbf{I}$ s.t. $cones(I) = cones(\mathbf{S}) \cap cones(\mathbf{T})$	GEN $\cap$ GEN = GEN	coNP	exponential
		IRR $\cap$ IRR = IRR	coNP	exponential
Minimization	$\mathsf{CST\_Minimization}(\mathbf{S}, \mathbf{T}) = \mathbf{D}$ s.t. $elem(\mathbf{D}) = \{\mathbf{m} \in elem(\mathbf{S}) \mid \mathbf{m} \notin cones(\mathbf{T})\}$	GEN $\div$ GEN = GEN	coNP	exponential
		IRR $\div$ IRR = IRR	coNP	exponential

Fig. 3. Complexity of the operations on CSTs.

IRR) CSTs, and operations that relax this conditions and accept *generic* (abbreviated as GEN) CSTs in input. The table in Figure 3 summarizes the complexity results presented in this section. Let us start from the simplest operations.

7.1 Emptiness Test

First of all, let us note that emptiness can be checked in constant time. In fact, it is sufficient to test if the CST contains only the *root* and *end* nodes.

7.2 Satisfiability Test

Checking that $\mathbf{t} \models \varphi$ on the CST $\mathbf{S}_\varphi$ has cost *linear* in the size of $\mathbf{S}_\varphi$, i.e., possibly logarithmic in the size of φ. In fact, the following theorem holds.

Theorem 4. *Let* $\mathbf{S}$ *be a CST and* $\mathbf{t}$ *be a vector of length* k*. We can check if there exists* $\mathbf{t}' \in elem(\mathbf{S})$ *such that* $\mathbf{t}' \preccurlyeq \mathbf{t}$*, namely* $\mathbf{t} \in cones(\mathbf{S})$*, in time linear in the number of edges of* $\mathbf{S}$*.*

Proof. We exhibit an algorithm that has time complexity that is linear in the size of $\mathbf{S}$. The algorithm is based on a *layer-by-layer* comparison of the labels of the nodes of $\mathbf{S}$ with the components of the vector: at a given layer, we mark all nodes that belong to a path (tuple) that is candidate to subsume the vector. Specifically, we first mark $root^{\mathbf{S}}$. Then, if a node m at the i-th layer of $\mathbf{S}$ is marked, we mark all successors n of m such that $t_{i+1} \geq val(n)$. At the end of the procedure, $\mathbf{t} \in cones(\mathbf{S})$ holds if and only if the node $end^{\mathbf{S}}$ is marked (note: we assume that $t_{k+1} = \bot$ and $\bot \geq \bot$). $\square$

7.3 Substitution

The substitution operation applied on a formula φ represented by a CST $\mathbf{S}_\varphi$ can be computed in linear time in the length of $\mathbf{S}_\varphi$.

Theorem 5. *Let* $\mathbf{S}_\varphi$ *be a CST. Then, we can compute a CST* $\mathbf{S}_{\varphi[x_i+c/x_i]}$ *such that* $elem(\mathbf{S}_{\varphi[x_i+c/x_i]}) = gen(\varphi[x_i + c/x_i])$ *in linear time.*

Proof. We build the CST $\mathbf{T} = \mathbf{S}_{\varphi[x_i+c/x_i]}$ as follows. $\mathbf{T}$ has the same components as $\mathbf{S}_\varphi$ except for the valuation function. Let val be the evaluation mapping of $\mathbf{S}_\varphi$ and val' be the evaluation mapping of $\mathbf{T}$. Then, $val'(n) = val(n) - c$ for every node $n \in N_i$, and $val'(m) = val(m)$ for $m \in N_j$ and $j \neq i$ (we use the same set of nodes for the two CSTs). $\square$

7.4 Subsumption

Let us first remark the following property. In any *irredundant* CST the constraints (atomic $\mathcal{U}$-formula) associated with a given variable (i.e. to the nodes of the corresponding layer) are not required to represent disjoint sets of tuples. In other words, we may look at a CST $\mathbf{S}$ as a sort of non-deterministic automata accepting the tuples of his *rich denotation*. Non-determinism is the reason why we often obtain a compact representation. Subsumption between two CSTs however has the complexity of *trace inclusion* for non-deterministic, acyclic automata. We first analyze the complexity of the problem for *irredundant* CSTs.

The case IRR $\subseteq$ *IRR.* In this case we have the following result.

Theorem 6. *The subsumption problem for two irredundant CSTs is* coNP*-complete in the size of the CSTs.*

Proof. Let $\mathbf{S}_1$ and $\mathbf{S}_2$ be irredundant CSTs. The problem is checking whether or not $cones(\mathbf{S}_1) \subseteq cones(\mathbf{S}_2)$. We can decide the complement of the problem by guessing a tuple of length k and testing in linear time (cf. Theorem 4) that it is subsumed by $\mathbf{S}_1$ and not by $\mathbf{S}_2$. Since CSTs encode $\mathcal{U}$-formulas, we can restrict our attention to a finite (exponential) set of candidate tuples using Lemma 1 as in the proof of Theorem 2.

To prove hardness, we use a reduction from *validity* as in Theorem 1. More precisely, given a propositional formula in DNF $F = D_1 \vee \ldots \vee D_n$ we define the CST $\mathbf{S}_{val(F)}$ that encodes all valuations that satisfy F. We reduce then validity to the subsumption test between $\mathbf{S}_{val(true)}$ and $\mathbf{S}_{val(F)}$.

Following the proof of Theorem 1, we first build the two $\mathcal{U}$-formulas $\Phi_{val(true)}$ and $\Phi_{val(F)}$ that encode the valuations that make *true* and F true, respectively. Consider the $\mathcal{U}$-formulas $\Phi_{val(D_i)}$ we used in the proof of Theorem 1 to represent all evaluations that satisfy a disjunct D_i of a *propositional formula* F in DNF. The transformation of $\Phi_{val(D_i)}$ to a disjunctive formula gives a formula φ_i exponential in the size of $\Phi_{val(D_i)}$. However, the representation of φ_i using a sharing tree $\mathbf{S}_{\varphi_i}$ is polynomial in the size of the *original* formula $\Phi_{val(D_i)}$, i.e., logarithmic in φ_i. In other words, we can build $\mathbf{S}_{\varphi_i}$ directly from $\Phi_{val(D_i)}$ (the formal proof is by induction on the structure of the formula). Following from the previous observations, we know that the CST $\mathbf{T}_{val(true)}$ (with root r_t and terminal node e_t) that encodes $\Phi_{val(true)}$, and the CST $\mathbf{S}_i$ (with root r_i and terminal node e_i) that encodes $\Phi_{val(D_i)}$ have size polynomial in F. It remains now to define the irredundant CST $\mathbf{S}_{val(F)}$. Note that, as extensively discussed in Section 7.6, the *union* operation of Figure 1 does not guarantee the result to be an irredundant CST. Thus, instead of using *union* we proceed as follows. By construction, every CST $\mathbf{S}_i$ is irredundant. Based on this fact, we first *merge* all $\mathbf{S}_i$'s into a single CST $\mathbf{S}_{val(F)}$ with root r and terminal node e such that $succ(r) = \{r_1, \ldots, r_n\}$, $succ(e_1) = \{e\}, \ldots, succ(e_n) = \{e\}$. To ensure that for all pairs of distinct tuples (paths) $\mathbf{t}, \mathbf{t}'$ of $\mathbf{S}_{val(F)}$ $\mathbf{t} \not\preccurlyeq \mathbf{t}'$ and $\mathbf{t}' \not\preccurlyeq \mathbf{t}$, we set

- $val(r_i) = i$;
- and $val(e_i) = 2n + 1 - i$ for $i : 1, \ldots, n$.

i.e., from left-to-right: increasing values for the r_i's; decreasing values for the e_i's. Clearly, the resulting CST is *irredundant* and has polynomial size in the size of F. Similarly, we define $\mathbf{S}_{val(true)}$ as the CST with root r' and terminal node e' obtained from $\mathbf{T}_{val(true)}$ setting $succ(r'_t) = \{r_t\}$ and $succ(e'_t) = \{e_t\}$. Finally, since the values of the nodes r'_t and e'_t should not influence the comparison of the paths of $\mathbf{S}_{val(true)}$ and $\mathbf{S}_{val(F)}$, we set $val(r_t) = val(e_t) = 2n + 1$ (a value strictly greater than the maximal value in $\mathbf{S}_{val(F)}$). We give an example of the reduction in Figure 4. Based on this construction, we obtain that

$$F \textit{ is valid iff } cones(\mathbf{S}_{val(true)}) \subseteq cones(\mathbf{S}_{val(F)}).$$

```
function CST_Subsumption(S, S' : CSTs) : boolean
   S'' ← CST_Minimization(S', S);
   return CST_Empty(S'');
endfunction
```

Fig. 5. Algorithm for subsumption.

Since the reduction is *polynomial*, we obtain that the subsumption problem for CSTs is coNP-complete (as for arbitrary $\mathcal{U}$-formulas). □

The cost of subsumption may be exponential in the number of edges of the input CSTs. The result follows from the fact that $\mathcal{U}$-formulas in disjunctive form can be represented compactly via CSTs.

The case $GEN \subseteq GEN$. ¿From Theorem 6, we obtain the following corollary.

Corollary 2. *The subsumption problem for two CSTs is* coNP*-complete in the size of the CSTs.*

Proof. Given $\mathbf{S}_1$ and $\mathbf{S}_2$, we want to check $cones(\mathbf{S}_1) \subseteq cones(\mathbf{S}_2)$.

We can decide the complement of the problem by guessing a tuple of length k and testing (following from Theorem 4, in linear time) that it is subsumed by $\mathbf{S}_1$ and not by $\mathbf{S}_2$. Since CSTs encode $\mathcal{U}$-formulas, we can restrict our attention to a finite (exponential) set of candidate tuples using Lemma 1 as in the proof of Theorem 2. Hardness follows from Theorem 6, in fact, the case $IRR \subseteq IRR$ can be reduced to the case $GEN \subseteq GEN$. □

Algorithm for Subsumption The subsumption test can be performed symbolically via the algorithm of Figure 5 based on the procedure CST_Minimization that we will discuss in Section 7.8.

7.5 Equivalence Test and Reduction in Normal Form

Given two CSTs $\mathbf{S}$ and $\mathbf{S}'$, we first study the complexity of the equivalence test, defined as $cones(\mathbf{S}) = cones(\mathbf{S}')$. The equivalence test is easier than *subsumption*. This will be evident when considering comparisons of irredundant CSTs.

The case $IRR \equiv IRR$.

Theorem 7. *Let $\mathbf{S}$ and $\mathbf{S}'$ be two* irredundant *CSTs. Then, equivalence of $\mathbf{S}$ and $\mathbf{S}'$ can be decided in* linear time.

Proof. ¿From Theorem 3, it follows that two irredundant CSTs are equivalent if and only if they are isomorphic. To check the latter condition, we can use a parallel top-down traversal. □

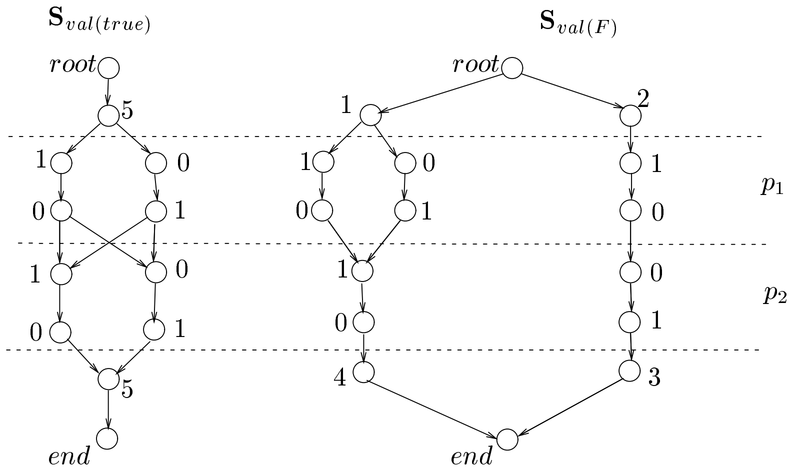

Fig. 4. The validity of $F = p_2 \vee (p_1 \wedge \neg p_2)$ is reduced to $cones(\mathbf{S}_{val(true)}) \subseteq cones(\mathbf{S}_{val(F)})$.

The case $GEN \equiv GEN$. The complexity of the equivalence test becomes harder, whenever one of the operands is not irredundant. As for subsumption, note that $GEN \equiv GEN$ cannot be reduced in polynomial time to $IRR \equiv IRR$.

Theorem 8. *Let $\mathbf{S}$ and $\mathbf{S}'$ be two CSTs. Then, equivalence of $\mathbf{S}$ and $\mathbf{S}'$ is* coNP-complete.

Proof. We can decide the complement of the problem by guessing a tuple of length k and testing in polynomial time that it is subsumed by $\mathbf{S}$, but not by $\mathbf{S}'$, or, vice versa, it is subsumed by $\mathbf{S}'$, but not by $\mathbf{S}$. Since CSTs encode $\mathcal{U}$-formulas, we can restrict our attention to a finite (exponential) set of candidate tuples using Lemma 1 as in the proof of Theorem 2.

To prove hardness, let us first note that $cones(\mathbf{S}) \subseteq cones(\mathbf{S}')$ if and only if $cones(\mathbf{S}) \cup cones(\mathbf{S}') = cones(\mathbf{S}')$. Furthermore, notice that a simple way to compute a generic CST $\mathbf{T}$ with $cones(\mathbf{T}) = cones(\mathbf{S}) \cup cones(\mathbf{S}')$ is to take $\mathbf{T}$ as the *sharing tree* union of $\mathbf{S}$ and $\mathbf{S}'$, i.e., $\mathbf{T} = \mathsf{union}(\mathbf{S}, \mathbf{S}')$. $\mathbf{T}$ will probably contain many redundant tuples, but this will not be a problem for our purposes. In fact, using the previous observation, and by the property of the union operation between sharing trees, we get a polynomial reduction from CSTs-subsumption (for generic CSTs) to CSTs-equivalence (for generic CSTs). From Corollary 2, it follows that checking equivalence between generic CSTs is coNP-hard. □

The previous result also applies to the case in which only one of the two CST is irredundant.

Corollary 3. *Let $\mathbf{S}$ be a generic CST, and $\mathbf{S}'$ be an irredundant CST. Checking $cones(\mathbf{S}) = cones(\mathbf{S}')$ is a* coNP-complete *problem.*

Proof. We use the same proof as for Theorem 8. □

As a consequence, we have the following result.

Corollary 4. *Given a CST $\mathbf{S}$, there is no* polynomial *algorithm that computes the irredundant CST $\mathbf{S}'$ such that $cones(\mathbf{S}) = cones(\mathbf{S}')$ (unless* P= NP*).*

Proof. Per absurdum, suppose we can use our algorithm as a polynomial-time oracle for reduction in normal form. Given the CSTs $\mathbf{S}$ and the irredundant CST $\mathbf{S}'$, we first use the *oracle* to compute in polynomial time the irredundant CST $\mathbf{T}$ such that $cones(\mathbf{S}) = cones(\mathbf{T})$. Then, we use Theorem 7 to decide in polynomial time if $cones(\mathbf{T}) = cones(\mathbf{S}')$. This way we have built a *polynomial* time algorithm that decides the problem of Corollary 3. This gives us a contradiction. □

Algorithm for Normalization The procedure of Figure 6 computes the normal form of a given CST $\mathbf{S}$ by first computing the CST **Sub** containing all the paths of $\mathbf{S}$ (strictly) subsumed by a distinct path. This is done by means of a parallel traversal defined in the procedure SubsumedPaths of Figure 6. As any path is subsumed by itself, we ensure that the parallel traversal compare only distinct paths using the flag *distinct*. The procedure memo! is used to cache intermediate results. The procedure memo? retrieves cached information or returns the *notmemorized* value if no information have ever been cached. Using these two procedures, we call the function SubsumedPaths at most once for each pair of nodes. As a consequence, SubsumedPaths has polynomial complexity. Finally, we use the sharing tree difference operation minus to removes all the paths of $\mathbf{S}$ that are also in **Sub**. To adjust condition 5 and 6 (that may be violated after running the procedure SubsumedPaths), we need to apply the algorithms of Appendix A. As the cost of adjusting condition 5 may be exponential, CST_NormalForm has a worst case exponential complexity.

```
function SubsumedPaths
        (n_S : node, n'_S : node, distinct : boolean) : node
  newdistinct ← distinct;
  if val^S(n_S) = ⊥ then
    if distinct = true then
      Sol ← new node such that val^Sub(Sol) = ⊥;
    else
      Sol ← null;
    endif
  else
    Sol ← new node s.t. val^Sub(Sol) = val^S(n_S);
    for each s_S ∈ succ^S(n_S), s'_S ∈ succ^S(n'_S) do
      if s_S ≠ s'_S then
        newdistinct ← true;
      endif
      if val^S(s_S) ≥ val^S(s'_S) then
        Son ← memo?(s_S, s'_S);
        if Son = notmemorized then
          Son ← SubsumedPaths(s_S, s'_S, newdistinct);
        endif
        if Son ≠ null then
          succ^Sub(Sol) ← succ^Sub(Sol) ∪ {Son};
        endif
      endif
    endfor
    if succ^Sub(Sol) = ∅ then
      Sol ← null;
    endif
  endif
  memo!(n_S, n'_S, Sol, newdistinct);
  return Sol;
endfunction
```

```
function CST_NormalForm(S : CST) : CST
  Let Sub be the CST built as follows
    root^Sub ← SubsumedPaths(root^S, root^S, false);
    Adjust_Condition5(Sub);
    Adjust_Condition6(Sub);
  return minus(S, Sub);
endfunction
```

Fig. 6. Algorithm that computes the normal form of a CST.

7.6 Disjunction

Given the formulas φ and ψ, we would like to symbolically compute their disjunction working over the corresponding CSTs $\mathbf{S}_\varphi$ and $\mathbf{S}_\psi$. Let us first consider the case in which the resulting CST needs not to be irredundant.

The case $GEN \cup GEN = GEN$. As already observed in the proof of Theorem 8, in this case we have an algorithm that works in polynomial time.

Theorem 9. *Let $\mathbf{S}$ and $\mathbf{S}'$ be two CSTs. Then, we can compute a CST $\mathbf{T}$ such that $cones(\mathbf{T}) = cones(\mathbf{S}) \cup cones(\mathbf{S}')$ by resorting to the operation of union of sharing trees $\mathbf{T} = \mathsf{union}(\mathbf{S}, \mathbf{S}')$, polynomial in the size of $\mathbf{S}$ and $\mathbf{S}'$ (see Figure 1).*

The case $IRR \cup IRR = IRR$. The problem becomes *harder* if we require the resulting CST to be *irredundant.*

Theorem 10. *Given two irredundant CSTs $\mathbf{S}$ and $\mathbf{S}'$, there is no* polynomial *algorithm that computes the irredundant CST $\mathbf{T}$ such that $cones(\mathbf{T}) = cones(\mathbf{S}) \cup cones(\mathbf{S}')$ (unless* P= NP*).*

Proof. Per absurdum, suppose we can use our algorithm as a polynomial-time oracle for computing the union of irredundant CSTs. Let us note that $cones(\mathbf{S}) \subseteq cones(\mathbf{S}')$ holds if and only if $cones(\mathbf{S}) \cup cones(\mathbf{S}') = cones(\mathbf{S}')$. In order to decide the subsumption problem $cones(\mathbf{S}) \subseteq cones(\mathbf{S}')$, we can first invoke the oracle to compute the irredundant CST $\mathbf{T}$ such that $cones(\mathbf{T}) = cones(\mathbf{S}) \cup cones(\mathbf{S}')$. Then, we use Theorem 7 to decide in polynomial time if $cones(\mathbf{T}) = cones(\mathbf{S}')$. This way we built a *polynomial* time algorithm to decide the *subsumption* problem for irredundant CSTs. This gives us the contradiction. □

The simplest way to compute disjunction is to first apply the polynomial algorithm for generic CSTs, and then put the resulting CST in normal form.

7.7 Conjunction

Given the formulas φ and ψ, we would like to symbolically compute their conjunction working over the corresponding CSTs $\mathbf{S}_\varphi$ and $\mathbf{S}_\psi$, i.e., compute a CST $\mathbf{T}$ such that $cones(\mathbf{T}) = cones(\mathbf{S}_\varphi) \cap cones(\mathbf{S}_\psi)$. Let us first study the case in which the operands are irredundant CSTs.

The case $IRR \cap IRR = IRR$. The following result shows that intersection is hard when the results is required to be irredundant.

Theorem 11. *Given two irredundant CSTs $\mathbf{S}$ and $\mathbf{S}'$, there is no* polynomial *algorithm that computes the irredundant CST $\mathbf{T}$ such that $cones(\mathbf{T}) = cones(\mathbf{S}) \cap cones(\mathbf{S}')$ (unless* P= NP*).*

Proof. Per absurdum, suppose we can use our algorithm as a polynomial-time oracle for computing the intersection of irredundant CSTs. Let us note that $cones(\mathbf{S}) \subseteq cones(\mathbf{S}')$ holds if and only if $cones(\mathbf{S}) \cap cones(\mathbf{S}') = cones(\mathbf{S})$. In order to decide the subsumption problem $cones(\mathbf{S}) \subseteq cones(\mathbf{S}')$, we can first invoke the oracle to compute the irredundant CST $\mathbf{T}$ such that $cones(\mathbf{T}) = cones(\mathbf{S}) \cap cones(\mathbf{S}')$. Then, we use Theorem 7 to decide in polynomial time if $cones(\mathbf{T}) = cones(\mathbf{S})$. This way we built a *polynomial* time algorithm to decide the *subsumption* problem for irredundant CSTs. This gives us the contradiction. □

The case $GEN \cap GEN = GEN$.

Theorem 12. *Given two CSTs* $\mathbf{S}$ *and* $\mathbf{S}'$*, there is no* polynomial *algorithm that computes the CST* $\mathbf{T}$ *such that* $cones(\mathbf{T}) = cones(\mathbf{S}) \cap cones(\mathbf{S}')$ *(unless* P= NP*).*

Proof. Following [27], we exhibit a polynomial *Turing* reduction from *validity*. Specifically, we build an OTM (Oracle Turing Machine) that uses *intersection* as an oracle to decide in polynomial time *validity*. To prove the validity of a propositional formula F in DNF, we first construct the CSTs $\mathbf{S}_{val(F)}$ and $\mathbf{S}_{val(true)}$, as shown in the proof of Theorem 6. This step can be done in *polynomial* time. By construction, we obtain then that F is *valid* if and only if $cones(\mathbf{S}_{val(true)}) \subseteq cones(\mathbf{S}_{val(F)})$, i.e., if and only if $cones(\mathbf{S}_{val(true)}) \subseteq cones(\mathbf{S}_{val(F)}) \cap cones(\mathbf{S}_{val(true)})$.

We apply then the oracle to compute the CST $\mathbf{T}$ such that $cones(\mathbf{T}) = cones(\mathbf{S}_{val(true)}) \cap cones(\mathbf{S}_{val(F)})$. Note that, by construction, the intersection of the cones induced by $\mathbf{S}_{val(true)}$ with those of $\mathbf{S}_{val(F)}$ will always returns cones whose minimal points have the labels of the minimal points of $\mathbf{S}_{val(true)}$ in the extra layers (the first and the last layer) used in the construction of Theorem 6. By hypothesis, $\mathbf{T}$ may contain redundant tuples. So we cannot apply a direct equivalence test between $\mathbf{T}$ and $\mathbf{S}_{val(true)}$ (as shown in Theorem 8, this test is exponential for generic operands). However, note that $\mathbf{S}_{val(true)}$ is irredundant by construction. Thus, to check $cones(\mathbf{S}_{val(true)}) \subseteq cones(\mathbf{T})$ we simply have to check that *at least* all generators of $\mathbf{S}_{val(true)}$ occur in $\mathbf{T}$, i.e., we have to check that $elem(\mathbf{S}_{val(true)}) \subseteq elem(\mathbf{T})$. This is the containment test between the flat denotation of *sharing trees* that we know to be polynomial (see Figure 1). Thus, the resulting Turing reduction is polynomial and coNP hardness for our search problem follows from the coNP completeness of *validity*. □

Algorithm for Conjunction Let us now describe the algorithm that we can use to compute the intersection between CSTs. Given the CSTs $\mathbf{S}$ and $\mathbf{S}'$, we compute $\mathbf{T}$ such that $cones(\mathbf{T}) = cones(\mathbf{S}) \cap cones(\mathbf{S}')$ as follows. We first define a pre-sharing tree $\mathbf{P}$ with the following components:

1. $N^{\mathbf{P}} = \{root\} \cup N_1^{\mathbf{P}} \cup \ldots \cup N_k^{\mathbf{P}} \cup \{end\}$ with $N_i^{\mathbf{P}} = \{(n,m) \mid n \in N_i^{\mathbf{S}}, m \in N_i^{\mathbf{S}'}\}$ for $i : 1 \ldots k$, i.e., a node in $\mathbf{P}$ correspond to a pair consisting of a node of $\mathbf{S}$ and a node of $\mathbf{S}'$ (of the same layer);
2. $val^{\mathbf{P}}((n,m)) = max(val^{\mathbf{S}}(n), val^{\mathbf{S}'}(m))$, and
3. we set $succ^{\mathbf{P}}(root) = N_1^{\mathbf{P}}$, for all $(n,m) \in N_1 \cup \ldots N_k : succ^{\mathbf{P}}((n,m)) = \{(n',m') | n' \in succ^{\mathbf{S}}(n), m' \in succ^{\mathbf{S}'}(m)\}$, and for all $(n,m) \in N_k^{\mathbf{P}} : succ^{\mathbf{P}}((n,m)) = \{end\}$.

We obtain the CST $\mathbf{T}$ from $\mathbf{P}$ by enforcing conditions (5-6) of Section 5 with the algorithms in Appendix A, Figure 33 and Figure 34. The algorithm to adjust condition 5 may have exponential time complexity in the size of the CST, whereas the time complexity to adjust condition 6 is polynomial. Note that $\mathbf{T}$ can contain redundancies, even if $\mathbf{S}$ and $\mathbf{S}'$ are irredundant. To get an irredundant CST, we have to transform $\mathbf{T}$ in normal form.

7.8 Minimization

Given the CSTs $\mathbf{S}_\varphi$ and $\mathbf{S}_\psi$ associated with the formulas φ and ψ, the operations of simplification returns the CST $\mathbf{S}_{\varphi \div \psi}$ such that $cones(\mathbf{S}_{\varphi \div \psi}) = [\![\varphi \div \psi]\!]$.

Theorem 13. *Given the CSTs* $\mathbf{S}_\varphi$ *and* $\mathbf{S}_\psi$*, there is no* polynomial *algorithm, that computes the CST* $\mathbf{S}_{\varphi \div \psi}$ *such that* $cones(\mathbf{S}_{\varphi \div \psi}) = [\![\varphi \div \psi]\!]$ *(unless* P= NP*).*

Proof. We exhibit a polynomial *Turing* reduction from *validity*. Specifically, we build an OTM (Oracle Turing Machine) that uses $\div$ as an oracle to decide in polynomial time *validity*. We first construct the two CSTs $\mathbf{S}_{val(F)}$ and $\mathbf{S}_{val(true)}$, as shown in the proof of Theorem 6.

We apply then the oracle on $\mathbf{S}_{val(true)}$ and $\mathbf{S}_{val(F)}$ to compute the CST $\mathbf{S}_{val(true) \div F}$. Finally we note that F is valid if and only if $\mathbf{T}$ is the empty CST. Checking emptiness can be done in constant time. Thus, the resulting Turing reduction is polynomial and coNP hardness for our search problem follows from the coNP completeness of *validity*. □

Algorithm for minimization The algorithm presented in Figure 7 computes the simplification operation for two CSTs $\mathbf{S}$ and $\mathbf{T}$. The procedure CST_Minimization of Figure 7 works similarly to the procedure of Figure 6. The function CST_Minimization first applies a procedure called SubsumedPathsthat computes the paths of $\mathbf{S}$ that are subsumed by at least one path in $\mathbf{T}$. The complexity of CST_Minimization is exponential in the size of $\mathbf{S}$ and $\mathbf{T}$ because SubsumedPaths returns a pre-CST that must be (potentially) determinized.

8 Simulation Relations and Related Heuristics

In the previous section we have proved several hardness results related to important problems defined on CSTs. In the section we introduce techniques that will give us sufficient and efficient conditions to recognize positive instance of the hard problems. More precisely, we will introduce *approximations* of the subsumption relation that can be tested more efficiently. The main idea is to exploit the *graph-structure* underlying CSTs, and in particular the possibility of defining *relations* between nodes of CSTs.

Technically, we will try to find sufficient condition for subsumption and equivalence by using *simulation relations*. Let $\mathbf{S}_1$ and $\mathbf{S}_2$ be two CSTs. Note that, if

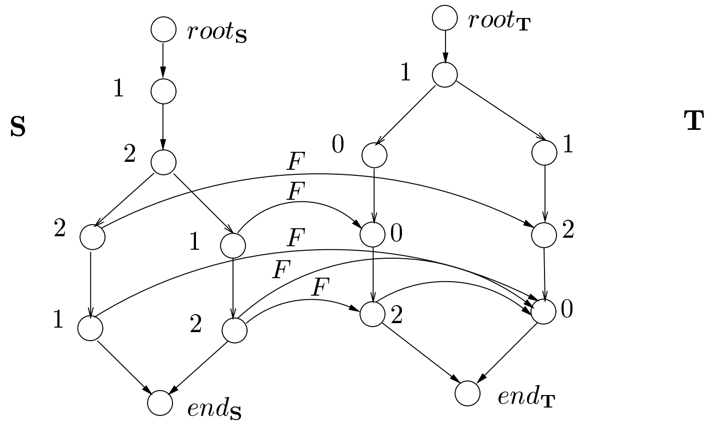

Fig. 8. The forward simulation is incomplete with respect to subsumption.

function SubsumedPaths($n_{\mathbf{S}}$: *node*, $n_{\mathbf{T}}$: *node*) : *node*
  if $val^{\mathbf{S}}(n_{\mathbf{S}}) = \bot$ **then**
    $Sol \leftarrow$ *new node such that* $val^{\mathbf{Sub}}(Sol) = \bot$;
  else
    $Sol \leftarrow$ *new node s.t.* $val^{\mathbf{Sub}}(Sol) = val^{\mathbf{S}}(n_{\mathbf{S}})$;
    for each $s_{\mathbf{S}} \in succ^{\mathbf{S}}(n_{\mathbf{S}}),\ s_{\mathbf{T}} \in succ^{\mathbf{T}}(n_{\mathbf{T}})$ **do**
      if $val^{\mathbf{S}}(s_{\mathbf{S}}) \geq val^{\mathbf{T}}(s_{\mathbf{T}})$ **then**
        $Son \leftarrow$ memo?($s_{\mathbf{S}}, s_{\mathbf{T}}$);
        if $Son = notmemorized$ **then**
          $Son \leftarrow$ SubsumedPaths($s_{\mathbf{S}}, s_{\mathbf{T}}$);
        endif
        if $Son \neq null$ **then**
          $succ^{\mathbf{Sub}}(Sol) \leftarrow succ^{\mathbf{Sub}}(Sol) \cup \{Son\}$;
        endif
      endif
    endfor
    if $succ^{\mathbf{Sub}}(Sol) = \emptyset$ **then**
      $Sol \leftarrow null$;
    endif
  endif
  memo!($n_{\mathbf{S}}, n_{\mathbf{T}}, Sol$);
  return *Sol*;
endfunction

function CST_Minimization($\mathbf{S}$: CST, $\mathbf{T}$: CST) : CST
  Let **Sub** be the CST built as follows
    $root^{\mathbf{Sub}} \leftarrow$ SubsumedPaths($root^{\mathbf{S}}, root^{\mathbf{T}}$);
    Adjust_Condition5(**Sub**);
    Adjust_Condition6(**Sub**);
  return minus($\mathbf{S}$, **Sub**);
endfunction

Fig. 7. Algorithm that minimizes a CST $\mathbf{S}$ with respect to a CST $\mathbf{T}$.

$elem(\mathbf{S}_1) \subseteq elem(\mathbf{S}_2)$ then $cones(\mathbf{S}_1) \subseteq cones(\mathbf{S}_2)$. Besides giving a sufficient condition for checking subsumption, the previous fact suggests a possible strategy to reduce the cost of the complete test. We first compute $\mathbf{T}$ = minus($\mathbf{S}_1, \mathbf{S}_2$) (polynomial in the size of $\mathbf{S}_1, \mathbf{S}_2$) and then test $cones(\mathbf{T}) \subseteq cones(\mathbf{S}_2)$ on the (possibly) smaller CST $\mathbf{T}$.

In the rest of this section we will give more interesting polynomial-time *sufficient conditions* for the subsumption test, based on a notion of *simulation* between nodes of CSTs. We will see that this notion of simulation is also useful to approximate both the reduction in normal form and the simplification operation on CSTs.

Given two nodes n and m of a CST $\mathbf{S}$ we are looking for a relation $\curvearrowright^F$ such that: $n \curvearrowright^F m$ implies $cones(\mathbf{S}_n) \subseteq cones(\mathbf{S}_m)$.

Definition 15 (Forward Simulation). Given two CSTs $\mathbf{S}$ and $\mathbf{T}$, let n be a node of the i-th layer of $\mathbf{S}$, and m be a node of the i-th layer of $\mathbf{T}$. We say that n is *forward simulated by* m, written $n \curvearrowright^F m$, if $val^{\mathbf{S}}(n) \geq val^{\mathbf{T}}(m)$ and for all the nodes $s \in succ^{\mathbf{S}}(n)$ there exists a node $t \in succ^{\mathbf{T}}(m)$ such that $s \curvearrowright^F t$.

Let $father^{\mathbf{S}}(n) = \{\ m \mid n \in succ^{\mathbf{S}}(m)\ \}$ be the set of fathers of a node n at layer i of a CST $\mathbf{S}$ ($fathers^{\mathbf{S}}(n) \subseteq N_{i-1}$). We define the backward simulation as follows:

Definition 16 (Backward Simulation). Given two CSTs $\mathbf{S}$ and $\mathbf{T}$, let n be a node of the i-th layer of $\mathbf{S}$, and m be a node of the i-th layer of $\mathbf{T}$. We say that n is *backward simulated by* m, written $n \curvearrowright^B m$, if $val^{\mathbf{S}}(n) \geq val^{\mathbf{T}}(m)$ and for all the nodes $s \in fathers^{\mathbf{S}}(n)$ there exists a node $t \in fathers^{\mathbf{T}}(m)$ such that $s \curvearrowright^B t$.

The following result (taken from [28]) shows that the previous simulations can be computed efficiently.

Theorem 14 (From [28]). Forward and backward simulation relations between the nodes of the CST $\mathbf{S}$ and the nodes of the CST $\mathbf{T}$ can be computed in $O(m \cdot n)$ where m is the sum of the number of nodes in $\mathbf{S}$ and in $\mathbf{T}$, and n is the sum of the number of edges in $\mathbf{S}$ and in $\mathbf{T}$.

8.1 Properties of the simulation

The following propositions relate subsumption and the simulation $\curvearrowright^F$.

Lemma 2. Given the CSTs $\mathbf{S}$ and $\mathbf{T}$, let $\mathbf{S}_n$ and $\mathbf{T}_m$ be the sub-CSTs rooted at nodes n and m, respectively. If $n \curvearrowright_F m$ then $cones(\mathbf{S}_n) \subseteq cones(\mathbf{S}_m)$.

Proof. We will show that for every node n of $\mathbf{S}$ in layer $k-l$, for $0 \le l \le k$ the property is verified. We reason by induction on the value of l.

Base case: $l = 0$. Let us consider $n \in N^{\mathbf{S}}_{k-l}$ and $m \in N^{\mathbf{T}}_{k-l}$ such that $n \curvearrowright_F m$. By definition of CSTs, we have that $succ^{\mathbf{S}}(n) = \{end\}$, $succ^{\mathbf{T}}(m) = \{end\}$, $elem(\mathbf{S}_{end}) = \emptyset$, and $elem(\mathbf{T}_{end}) = \emptyset$. Thus, $elem(\mathbf{S}_n) = \langle val^{\mathbf{S}}(n)\rangle$ and $elem(\mathbf{T}_m) = \langle val^{\mathbf{T}}(m)\rangle$, and, by definition of $\curvearrowright_F$, we have that $val^{\mathbf{S}}(n) \ge val^{\mathbf{T}}(m)$.

Induction step: $l > 0$. By induction hypothesis, for all nodes $n' \in N^{\mathbf{S}}_{k-i}$and $m' \in N^{\mathbf{T}}_{k-i}$, with $0 \le i < l$, we know that if $n' \curvearrowright_F m'$ then $cones(\mathbf{S}_{n'}) \subseteq cones(\mathbf{S}_{m'})$. Let us now consider $n \in N^{\mathbf{S}}_{k-l}$ and $m \in N^{\mathbf{T}}_{k-l}$ with $n \curvearrowright_F m$. By definition of $\curvearrowright_F$, we know that for $s \in succ^{\mathbf{S}}(n)$ there exists $t \in succ^{\mathbf{T}}(m)$ such that $s \curvearrowright_F t$. We also know that $val^{\mathbf{S}}(n) \ge val^{\mathbf{T}}(m)$. Thus, we have that

$$elem(\mathbf{S}_n) = \{\{val^{\mathbf{S}}(n)\} \times elem(\mathbf{S}_{n'}) \mid n' \in succ^{\mathbf{S}}(n)\}$$

and

$$elem(\mathbf{T}_m) = \{\{val^{\mathbf{T}}(m)\} \times elem(\mathbf{T}_{m'}) \mid n' \in succ^{\mathbf{T}}(m)\}.$$

Thus, it follows that $cones(\mathbf{S}_n) \subseteq cones(\mathbf{T}_m)$ holds. □

The converse does not hold (in accord with the coNP hardness result for subsumption). As a counterexample, take the two CSTs in Figure 8. The curly arrows represent the simulation relation between nodes of $\mathbf{S}$ and $\mathbf{T}$. Note that none of the nodes of layer 2 in $\mathbf{T}$ simulates the single node of $\mathbf{S}$ at the same layer. However, the denotation of $\mathbf{S}$ are contained in that of $\mathbf{T}$. In fact, $\langle 1,1,2,0\rangle \preccurlyeq \langle 1,2,2,1\rangle$ and $\langle 1,0,0,2\rangle \preccurlyeq \langle 1,2,1,2\rangle$.

As Lemma 2 can be reformulated for backward simulation, the following theorem holds.

Theorem 15. Let $root^{\mathbf{S}}, end^{\mathbf{S}}$ and $root^{\mathbf{T}}, end^{\mathbf{T}}$ be the root and terminal nodes of $\mathbf{S}$ and $\mathbf{T}$, respectively. If $root^{\mathbf{S}} \curvearrowright_F root^{\mathbf{T}}$ then $cones(\mathbf{S}) \subseteq cones(\mathbf{T})$. Symmetrically, if $end^{\mathbf{S}} \curvearrowright_B end^{\mathbf{T}}$ then $cones(\mathbf{S}) \subseteq cones(\mathbf{T})$.

The previous theorem gives us sufficient conditions for testing subsumption.

8.2 Use of simulations to remove redundancies

As for the subsumption test, the simulations we introduced in Section 8 can be used to approximate the exact normalization procedure and the exact simplification procedure. For this purpose, we introduce a rule that allows us to exploit the information given from the simulation relations in order to locally remove edges of a CST.

Definition 17 (Edge Removal). Given a CST $\mathbf{S}$ having two nodes $n, m \in N^{\mathbf{S}}$ with $n \curvearrowright_B m$ such that there exist $s \in succ^{\mathbf{S}}(n)$ and $t \in succ^{\mathbf{S}}(m)$ $(s \ne t \vee n \ne m)$ with $s \curvearrowright_F t$, we define $remove(\mathbf{S}, \langle n, s\rangle)$ as the pre-CST with successor relation $succ'$ obtained from $\mathbf{S}$ by setting $succ'(n) = succ(n) \setminus \{s\}$.

The following proposition states the correctness of the previous rule. Its correctness follows directly from the definition of *remove*.

Proposition 3. *Given a CST* $\mathbf{S}$ *with the nodes* m, n *with* $n \curvearrowright_B m$ *such that there exist* $s \in succ^{\mathbf{S}}(n)$ *and* $t \in succ^{\mathbf{S}}(m)$ $(s \ne t \vee n \ne m)$ *with* $s \curvearrowright_F t$*, then*

$$cones(\mathbf{S}) = cones(remove(\mathbf{S}, \langle n, s\rangle)).$$

A possible strategy to apply Definition 17 consists of the on-the-fly removal of edges during the computation of the simulation relations. Specifically, during a bottom-up traversal of the CST (assume the backward simulation is already computed), we first apply the rule of definition 17 to every pairs of edges between nodes of layers N_i and N_{i+1}. Removing edges between nodes of these layers does not change the structure of the CST in the set of layers $S_1 = \{\ N_j$ such that $j \le i\}$ and the set of layers $S_2 = \{N_j$ such that $j \ge i+1\ \}$. As a consequence, the backward simulation does not change for the nodes of the layers in S_1 and the forward simulation does not change for the nodes of the layers in S_2. This allow us to propagate the application of the rule bottom-up. The rule to remove edges is applied exhaustively at each step.

Note that after the bottom-up traversal, the forward simulation is computed on the nodes of the resulting pre-CST but the backward simulation could have changed. An example is given in Figure 10, where B and F represent backward and forward simulation, respectively. Since $3 \curvearrowright_B 2$ and $6 \curvearrowright_F 5$, we can remove the arc $3 \to 6$ during the bottom-up traversal, doing so we will add $\langle 6, 0\rangle$ to $\curvearrowright_B$. By doing a top-down traversal, some new edges should be removed again. As a consequence, we get better and better results iterating bottom-up and top-down traversals. Also note that the pre-CST $remove(\mathbf{S}, \langle n, s\rangle)$ may violate the condition 6 of the definition of sharing trees (Section 5). For instance, removing the node with label 6 from the CST in Figure 9 makes the two nodes with label 2 (belonging to vectors that are not in the subsumption relation) have the same set of successors. As already mentioned in the course of the paper, condition 6 can be restored using the algorithm in Appendix A, Figure 34, polynomial in the size of the CST.

To approximate the simplification procedure, we can define a definition similar to definition 17 where the nodes s, n and the nodes t, m are in two different CSTs.

A simplified version of the rule of definition 17 that requires only a local test for every node of a CST is given as follows.

Definition 18 (Local Edge Removal). Given the CST $\mathbf{S}$ with the nodes $n, m, s, t \in N^{\mathbf{S}}$ with $s \in succ^{\mathbf{S}}(n)$

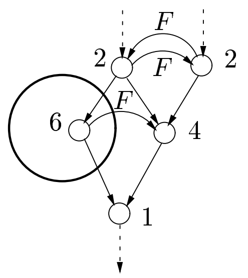

Fig. 9. Violation of condition 6 after removing two edges and one node.

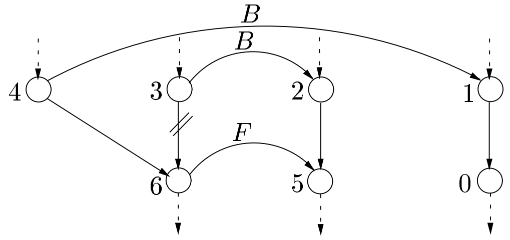

Fig. 10. Removing edge $3 \to 6$ changes the backward simulation.

and $t \in succ^{\mathbf{S}}(m)$ $(s \neq t \vee n \neq m)$ such that $val^{\mathbf{S}}(n) \geq val^{\mathbf{S}}(m)$, $val^{\mathbf{S}}(s) \geq val^{\mathbf{S}}(t)$,$succ^{\mathbf{S}}(s) \subseteq succ^{\mathbf{S}}(t)$ and $father^{\mathbf{S}}(n) \subseteq father^{\mathbf{S}}(m)$. Then, $local\_remove(\mathbf{S}, \langle n, s\rangle)$ is the pre-CST in which the successor relation $succ'$ is obtained from $\mathbf{S}$ by setting $succ'(n) = succ^{\mathbf{S}}(n) \setminus \{s\}$.

Though less effective than the rule of definition 17, the rule of definition 18 can be paired with it in order to simplify the computation of the simulation relations.

9 Application to Parameterized Verification

The theory of well-structured systems [1,21] gives us decision procedures to verify a large class of safety properties of *parameterized* systems modeled as Petri Nets [1,21], Lossy Vector Addition Systems [5], and Broadcast Protocols [17]. The decision procedures are based on *backward reachability* algorithms like the one proposed in [1], whose termination (for Petri Nets and their monotonic extensions) is guaranteed by Dickson's Lemma [13]. It is important to recall that forward approaches like Karp-Miller's coverability tree [25] are not robust when applied to extensions of Petri Nets like Broadcast protocols [17].

Differently from the finite-state case, in parameterized verification *symbolic* representations are ineluctable in order to make the approach *effective*: we *need* to finitely represent infinite collections of states. We have to deal with upward-closed sets of markings when we restrict our attention to Petri Nets. Two examples of symbolic representations for *upward-closed sets* of marking are collections of minimal points [1], and *linear arithmetic constraints* [12]. The complexity of the algorithm of [1] is non-elementary. For this reason, naive implementations of the backward approach suffer from the *symbolic state explosion* problem: the number of minimal points or, equivalently, the size of the constraints become unmanageable after few iterations. Symbolic state explosion is the counterpart of the state explosion problem we known from finite-state verification.

In this section we will focus our attention on the application of CSTs to verification problems of Petri Nets. As explained in the first part of the paper, CSTs can be used to compactly store the minimal points of an upward-closed set of markings. Using this symbolic representation, in the preliminary results given in [14] we managed to prove properties of Petri Nets (of small size) that could not be managed from other infinite-state model checkers (working backward) like HyTech [29].

In order to attack *symbolic state explosion* for Petri Nets of large size, we will combine our CST-based technology with heuristics based on the *structural properties* of Petri Nets. More precisely, the heuristic rule we propose in this paper is based on a *fully automatic* static analysis [34], whose results can be used during the backward reachability algorithm to significantly cut the search space. As the other general purpose heuristics presented in the first part of the paper (e.g. simulation relations), our *structural heuristic* works in polynomial time on the graph structure of CSTs. When combined with our CSTs-symbolic representation, the heuristic rule allow us to scale up the dimension of the case-studies of one order of magnitude.

Before entering in more details, in the next section we will briefly recall the main ideas behind the connection between parameterized systems, Petri Nets, and backward reachability.

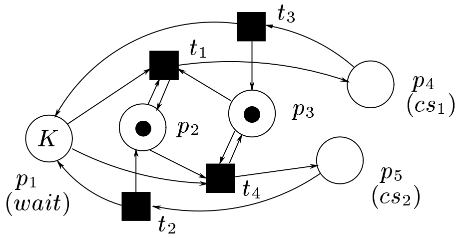

Fig. 11. An example of Petri Net with parametric initial marking $\langle K, 1, 1, 0, 0\rangle$, $K \geq 1$.

9.1 Petri Nets and Verification of Safety Properties

Following [23], asynchronous concurrent systems can be naturally represented as Petri Nets in which places and transitions are used to model *local states*, *internal actions* and communication via *rendez-vous* (see e.g. [33, 34] for an introduction to Petri Nets). At this level of abstraction, processes can be viewed as indistinguishable black *tokens*. A *marking* $\mathbf{m} = \langle m_1, \ldots, m_n\rangle$, a mapping from places to *non-negative* integers, can be viewed as an abstraction of a *global system state* in which we only keep track of the number of processes in every state. The number of processes in the system is determined by the *initial marking* $\mathbf{m_0}$. The *backward reachability* approach is based on the property of the relation $\preccurlyeq$ of being a *well-quasi ordering*. This property ensures the termination of backward reachability, whenever the starting point of the exploration is an upward-closed set of markings [19, 1,21].

As an example, consider the Petri Net of Figure 11, a monitor for a parameterized system with two mutually exclusive critical sections (cs_1 and cs_2). Initially, all K processes are in p_1. To enter cs_1, a process tests for the presence of processes in cs_2 using p_2, and locks cs_1 using p_3 (transition t_1), and vice versa. Processes leave the critical section using transitions t_3 and t_4. Note that the set U of violations to mutual exclusion is the *upward-closed* set generated by the *minimal violations* $\langle 0,0,0,2,0\rangle$, $\langle 0,0,0,0,2\rangle$, and $\langle 0,0,0,1,1\rangle$ (*at least* 2 tokens in p_4+p_5). To prove that the protocol guarantees *mutual exclusion* for *any* value of K, it is enough to show that no admissible initial marking is in the set of *predecessor* markings $pre^*(U)$ of U (*pre* is the operator that returns the set of markings that reach some marking in U by *firing* a transition). To compute $pre^*(U)$, we iterate the application of the predecessor operator *pre* until we reach a fixpoint. During the computation, every newly generated marking is stored only if it is *not subsumed* by an already visited one. The backward reachability graph of our example is given in Figure 12. In Figure 12 we have omitted all redundant markings (about 30). As mentioned in the introduction, the symbolic backward approach based on the enumeration of minimal points of sets of markings suffers from the symbolic state explosion problem. More sophisticated data structures are necessary to make the approach feasible in practice.

9.2 Symbolic Verification of Parameterized Systems

Following, e.g. [33,34], a Petri Net can be represented via a *vector addition system*, i.e., a machine model that consists of n variables $x_1, \ldots, x_n$ (one for each place) ranging over non-negative integers, and m *guarded commands* (one for each transition) defined over the data variables. The guarded command associate with the i-th transition contains a guard $x_j \geq c_{ij}$ and an assignment $x_j := x_j + d_{ij}$ for every place p_j. The guard is such that

- $c_{ij} \geq 0$ if $d_{ij} \geq 0$: when reasoning backward, it avoids to assign negative values to x_j;
- $c_{ij} \geq d_{ij}$ if $d_{ij} < 0$: when reasoning forward, it avoids to assign negative values to x_j.

Markings represent then the current state of the n-counters associated with the places. Having this view in mind, it should be very easy to see how to apply $\mathcal{U}$-formulas for representing *covering* problems of Petri Nets, and thus how to use CSTs to efficiently solve them.

Given a Petri Net N and an upward-closed set of markings U, we will show that we can define a $\mathcal{U}$-formula to represent the set of *predecessors* of markings in U with respect to the transitions in N. Actually, we already described in Section 3 how to represent an upward-closed set U as a $\mathcal{U}$-formula φ_U. It remains us to show how to construct the $\mathcal{U}$-formula that represents $pre(U)$.

Definition 19 (The $\mathcal{U}$-formula for $pre(U)$). Let $\varphi_i = \wedge_{j=1\ldots n}\, x_j \geq c_{ij}$ be the guard of the i-th guarded command (transition) and let $x_j := x_j + d_{ij}$ be the action associated with the i-th guarded command (transition) and to the jth variable (place). The following $\mathcal{U}$-formula can be used to represent $pre(U)$:

$$pre(\Phi_U) = \bigvee_{i=1}^{m} (\varphi_i \wedge \Psi_i)$$

where $\Psi_i = \Phi_U[x_1 + d_{i1}/x_1] \ldots [x_n + d_{in}/x_n]$

It is easy to verify that $[\![\Phi_{pre(U)}]\!] = pre(U)$. Notice that according to results of section 3, if Φ is in disjunctive form we can use the specific algorithms on $\mathcal{U}$-formula

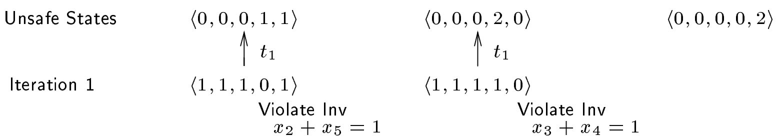

Fig. 12. Backward reachability graph.

function $\mathsf{BS}(\varphi_I, \varphi_U : \mathcal{U}-formulas) : boolean$
  $\Phi \leftarrow \varphi_U;$
  $\Psi \leftarrow empty; (i.e. [\![\Psi]\!] = \emptyset)$
  while $\Phi \not\models \Psi$ **do**
    $\Psi \leftarrow \Phi;$
    $\Phi \leftarrow \Psi \vee pre(\Psi); (pre(\Psi)\ as\ in\ Def.19)$
  endwhile
  return $([\![\varphi_I \wedge \Phi]\!] \not\equiv \emptyset);$
endfunction

Fig. 13. Backward reachability algorithm using $\mathcal{U}$-formulas.

in disjunctive form to obtain a formula $\Phi_{pre(U)}$ that is still in disjunctive form. In the rest of this section, we always use $\mathcal{U}$-formula in disjunctive form. Based on this idea, in order to verify problems that can be expressed via upward-closed sets, we can build the symbolic backward reachability algorithm shown in Figure 13. The algorithm works using $\mathcal{U}$-formulas and their operations. By construction, the intermediate results are always in disjunctive form. At each step of the algorithm, we use subsumption as termination test. Interestingly, this algorithm coincides with the backward search scheme implemented by symbolic model checkers HyTech [29] and ALV [7]. These tools use *linear arithmetic constraints* as assertional languages to represent *infinite* collection of system states (i.e. assignments of reals or integers to a given number of data variables). Furthermore, they internally maintain formulas in disjunctive form (linear arithmetic constraints are defined as conjunctions of atomic formulas). Thus, they can be used as effective implementation platform for the backward search algorithm based on $\mathcal{U}$-formulas described above. In [12], we tested this technology on some Petri Net example of small size. We report on these experiments in the following section.

9.3 Symbolic State Explosion

The *table manufacturing system* of Teruel [35] is a production system modeled via a *weighted Petri Net* with 7 places and 6 transitions. The values of the weights on the transitions vary from 2 to 4. This Petri Net can be modeled using a VAS with 13 variables, 7 for the places and 6 extra variables, say $T_1, \ldots, T_6$, to keep track of the number of firings of each transition. As shown in [35], all deadlock-free initial markings are exactly the predecessors of the set of states where the first transition is fired at least 3 times, and all the others are fired at least twice (i.e. $U = T_1 \geq 3 \ \wedge \ \bigwedge_{j>1} T_j \geq 2$). In [12], different general purpose constraint-based checkers have been used to compute the set of predecessors via backward reachability. As shown in Figure 14, the comparison was carried out using HyTech, the tool based on the Omega library of Bultan, Gerber and Pugh's [7], and a specialized implementation proposed in [12], in which collection of linear constraints are stored as a list of list of bit-vectors. As shown in the table of Figure 14, the general purpose model checkers suffer from the symbolic state explosion, due to the way in which constraints are stored, i.e., maintaining them in disjunctive normal form.

The specialized data structures proposed in [12] perform instead reasonably well. The data structure of [12] are based on the following idea. A constraint like $x \geq c$ is represented via a pair $\langle b, c\rangle$ in which b is a bit vector. Variable x_i is represented as the unit bit vector with 1 in position i (for n variables, we need then n bits). A conjunction of constraints can be stored then as a *list of such pairs*. Finally, a collection of constraint is stored as a *list of list of pairs*. As an example, the $\mathcal{U}$-formula in disjunctive normal form over the three variables $\langle x_1, x_2, x_3\rangle$ $\varphi_1 \vee \varphi_2$, where $\varphi_1 = (x_1 \geq 2 \wedge x_2 \geq 1 \wedge x_3 \geq 4)$ and $\varphi_2 = (x_1 \geq 2 \wedge x_2 \geq 4 \wedge x_3 \geq 1)$ is represented as following.

φ_1 :	$\langle 100, 2\rangle$	$\langle 010, 1\rangle$	$\langle 001, 4\rangle$
φ_2 :	$\langle 100, 2\rangle$	$\langle 010, 4\rangle$	$\langle 100, 1\rangle$

As shown in [12], bit-a-bit operations can be used to handle the resulting representation. Although when applied to $\mathcal{U}$-formulas, the use of bit-vectors to represent variables might seem a waste of space, this representation can be applied to capture more sophisticated classes of constraints (see [12]). As an example, we can represent constraints like $x_1 + x_2 + x_3 \geq 5$ via the pair $\langle 111, 5\rangle$.

Unfortunately, this symbolic representation of $\mathcal{U}$-formulas suffers from another potential source of redundancy. Note, in fact, that the $\mathcal{U}$-formula of our example can be written more compactly as $(x_1 \geq 2) \wedge ((x_2 \geq 1 \wedge x_3 \geq 4) \vee (x_2 \geq 4 \wedge x_3 \geq 1))$. Intuitively, what is missing in the data structure of [12] is the possibility of *sharing* information among different disjuncts. This justifies the use of CSTs as heuristics to obtain compact data structures for this kind of verification problems.

Examples	No.Rules	Unsafe States	Steps	BitVector[1]	HyTech[1]	Presb[2]
Man. System [35]	6	$T_1 \geq 3,\ \wedge_{i>1} T_i \geq 2$	24	1090s	6925.27s	19h50m

[1] On a Sun Sparc 5.6. [2] On a Sun Ultra Sparc.

Fig. 14. Comparisons of Constraint-based verification tools.

9.4 CST-based Symbolic Backward Reachability

The first step toward a CST-based symbolic backward search algorithm consists in formulating the $\mathcal{U}$-based algorithm using the corresponding CST-operations. After this preliminary step, we will try to exploit the structure of CSTs in order to find optimizations one can use during backward search. As mentioned in the previous sections, it is often convenient to work on generic CSTs (i.e. in which we allow redundancies). As an example, in this case disjunction can be done in polynomial time. On the other hand, too many redundancies might cause an explosion of the space needed to store intermediate results. The trade off between complexity and compactness of the CSTs is a crucial aspect to take into account, e.g., in the computation of the CST-based symbolic predecessor operator. Another important point to consider is the complexity of the subsumption test used as a termination test for the fixpoint computation of pre^*. In the next sections we will discuss these two points and then present the resulting CST-based backward search algorithm. Let us start discussing the symbolic computation of *pre*.

9.5 A CST-based CST_Pre *operation for Petri Nets*

Given a CST $\mathbf{S}_\varphi$ representing a $\mathcal{U}$-formula in disjunctive formula φ (hence an upward-closed set of markings), we would like to symbolically compute $pre(\varphi)$ using operations on CSTs. In the following we will focus our attention on the backward effect of a single Petri Net transition t, represented via a guarded command in which the guard $\varphi_{tj} = x_j \geq c_{ij}$ and the action $x_j := x_j + d_{tj}$ are associated with the j-th place. Let $\mathbf{S}_\varphi$ be a CST, then the operator $\mathsf{CST\_Pre}(t, \mathbf{S})$ should satisfy the following property

$$cones(\mathsf{CST\_Pre}(t, \mathbf{S}_\varphi)) = cones(\mathbf{S}_\psi),$$

where $\psi = \varphi_t \ \wedge\ (\varphi[x_1 + d_{i1}/x_1] \ldots [x_k + d_{ik}/x_k])$, and $\varphi_t = \varphi_{t1} \wedge \ldots \wedge \varphi_{tn}$ is the guard of the transition t. A possible implementation of this operation makes use of CSTs symbolic conjunction as follows:

$$\mathbf{S}_\psi = \mathsf{CST\_Intersection}(\mathbf{S}_{\varphi_t}, \mathbf{S}_{\varphi[x_1+d_{i1}/x_1]\ldots[x_k+d_{ik}/x_k]}).$$

The CST $\mathbf{S}_{\varphi[x_1+d_{i1}/x_1]\ldots[x_k+d_{ik}/x_k]}$ represents in a compact way the disjunction of constraints obtained after the backward application of the action of t. The final result is obtained by intersecting the resulting CST with the CST associated with the guard of t. Although the CST $\mathbf{S}_{\varphi_t}$ is a very special CST, i.e. it contains only one path, intersection makes this operation exponential in the size of $\mathbf{S}$. Actually, this is also the cost of search problem of symbolically computing $\mathsf{CST\_Pre}$ on CSTs. We will formally prove this claim in the following theorem.

Theorem 16. *Let* $\mathbf{S}$ *be an (irredundant) CST and* t *be a Petri Net transition. Then, there is no polynomial algorithm to compute the (irredundant) CST* $\mathbf{T}$ *such that* $cones(\mathbf{T}) = cones(\mathsf{CST\_Pre}(t, \mathbf{S}))$ *(unless* $\mathsf{P} = \mathsf{NP}$*).*

Proof. Let us first consider the case in which the input CST $\mathbf{S}$ is irredundant and the result is required to be irredundant, and then extend the proof to generic CSTs.

The case $\mathsf{CST\_Pre}(IRR) = IRR$. As in Section 6, we exhibit a polynomial *Turing* reduction from *validity*. Specifically, we build an OTM (Oracle Turing Machine) that uses $\mathsf{CST\_Pre}$ as an oracle to decide in polynomial time *validity*.

1. Let $F = D_1 \vee \ldots \vee D_n$ be a propositional formula in DNF defined over k symbols. We first construct the CST $\mathbf{T}_{val(D_i)}$ for $i : 1, \ldots, n$ and $\mathbf{T}_{val(true)}$, i.e. the CSTs without the extra layers used in the proof of Theorem 6. This step can be done in *polynomial* time.
2. As a second step, we add two extra layers on the top of every newly computed CST $\mathbf{T}_{val(D_i)}$; let us call $\mathbf{T}_i$ the so obtained CST. The labels in the two extra layers are such that all resulting $\mathbf{T}_i$'s have generators that are not comparable to each other. Specifically, we simply add the prefix with labels $\langle i, n - i + 1\rangle$ to every tuple in $\mathbf{T}_i$. Note that for each $\mathbf{T}_i$ we need only two nodes and polynomially many edges distributed in the two extra layers: the node in the first layer has label i, the node in the second layer has label $n - i + 1$; and all paths going out from the root are forced to pass through the two nodes.
3. Now, we build the CST $\mathbf{T}_F$ by first merging all the $\mathbf{T}_i$'s in a single pre-sharing tree, and then reducing it in a CST by sharing all their *suffixes* using the polynomial time algorithm proposed in [36] (see Appendix A). From now on, we will interpret the resulting CST $\mathbf{T}_F$ as representing markings for the variables $x, y, x_1, \ldots, x_k$, where x, y are the variables associated with the two newly introduced extra layers, and $x_1, \ldots, x_k$ are the variables associated with the propositional symbols in F. Note that n is the largest value occurring in the constraints associated with x, y is $\mathbf{T}_F$.

4. We define now a Petri Net transition t having the following guard:

$$x \geq n \ \wedge y \geq n \ \wedge x_1 \geq 0 \ \wedge \ \ldots \ \wedge x_k \geq 0.$$

and the following action

$$x' = x \ \wedge \ y' = y \ \wedge x_1' = x_1 \wedge \ldots \wedge x_k' = x_k.$$

This transition first removes and then put back n tokens from x and y. Thus at least n tokens must be in x and y in order for t to be fired. The transition has no effects on the other variables.
5. Notice that the result of computing $\mathsf{CST\_Pre}$ over the constraints associated with x and y in $\mathbf{T}_F$ is always the constraint $x \geq n \wedge y \geq n$ (by construction, the constants in $\mathbf{T}_F$ associated with x and y range in $1, \ldots, n$). As a consequence, if we apply the oracle to t and $\mathbf{T}_F$, we will obtain an irredundant CST in which: (1) there is only one path associated with the variables x and y with labels $\langle n, n \rangle$; (2) every evaluation for the propositional symbol of F occurs at most once. Point (2) is a consequence of (1) and of the fact that the oracle returns an irredundant CST.
6. We build now the CST $\mathbf{T}_{val(true)}$ associated with all possible evaluations of the propositional symbols in F, and then we add the prefix $\langle n, n \rangle$ to the tuples in $\mathbf{T}_{val(true)}$ as for the $\mathbf{T}_i$'s. Let $\mathbf{S}_{true}$ be the resulting CST. By construction, we obtain then that F is *valid* if and only if $cones(\mathsf{CST\_Pre}(t, \mathbf{T}_F)) = cones(\mathbf{S}_{true})$. Finally, we note that, since both CST are irredundant, we can compute their equivalence in polynomial time (see Theorem 3).

The resulting Turing reduction is polynomial, thus coNP hardness for our search problem follows from the coNP completeness of *validity.*

The case $\mathsf{CST\_Pre}(GEN) = GEN$. We can extend the proof of coNP hardness when $\mathsf{CST\_Pre}$ works on *generic* CSTs as follows. Instead of labeling the extra layers of the CST $\mathbf{T}_i$ using the prefix $\langle i, n-i+1 \rangle$, we always use the prefix $\langle i, i \rangle$. As a consequence, we will get a CST $\mathbf{T}_F$ that may contain redundant tuples. If now we apply the oracle (working on any CST) we will obtain a CST $\mathsf{CST\_Pre}(t, \mathbf{T}_F)$ that may also contain redundant tuples. Now, we build $\mathbf{S}_{true}$ exactly as is the case $\mathsf{CST\_Pre}(IRR) = IRR$. By construction, however, we notice that F is *valid* if and only if $cones(\mathsf{CST\_Pre}(t, \mathbf{T}_F))$ contains all generators of $cones(\mathbf{S}_{true})$. Following from Figure 1, we can check this property by using containment of the flat denotations $elem(\mathbf{S}_{true}) \subseteq elem(\mathbf{T}_F)$ in polynomial time. The resulting Turing reduction is again polynomial. Thus, coNP hardness for our search problem follows from the coNP completeness of *validity.* □

As shown in the previous theorem, the problems of computing $\mathsf{CST\_Pre}$ arise from the presence of guards in a Petri Net transition. Note, in fact, that if counters range over negative values that the computation becomes trivially polynomial (it amounts to a substitution operation). Nevertheless, we will show later that in practice computing $\mathsf{CST\_Pre}$ is never a bottleneck for the CST-based search engine. Intuitively, the reason is that in practice only few labels collapse to the same value. In most of the cases the application of $\mathsf{CST\_Pre}$ amounts to a substitution operation.

9.6 CST-driven Optimizations of CST_Pre

Following the definition of $\mathsf{CST\_Pre}$, we directly get the following symbolic algorithm to compute the pre-image of a set of markings:

$$\mathsf{CST\_Pre}(\mathbf{S}) = \cup_{t \in T} \ \mathsf{CST\_Pre}(t, \mathbf{S})$$

where T is the set of transitions of our Petri Net. In this section, we will focus on some optimization of this algorithm. In the following, we will assume that a transition t consists of

- a guard $x_i \geq d_{ti}$;
- and an action $x_i' = x_i + c_{ti}$

for each variable x_i associated with the i-th place, and stored in the i-th layer of a CST. We will also extend the mapping $cones(\cdot)$ to denote the rich denotation of a pre-CST (i.e. in which condition 5 or 6 may not hold). First, we can note that it is easy to characterize the set of predecessors of a upward-closed set with one minimal marking, as described by the following proposition.

Proposition 4. *Let* $\mathbf{m}$ *be a marking and* t *be a transition. Then, we have* $pre(t, cone(\mathbf{m})) = cone(\mathbf{m}')$ *where* $\mathbf{m}'(i) = max(g_{ti}, \mathbf{m}(i) - c_{ti})$

Following Proposition 4, we can compute the minimal marking of the predecessors set for a transition t via independent computations on each components. As the denotation of a CST is the union of the upward-closed set described by its paths, we can define an algorithm based on this idea. More precisely, given a CST $\mathbf{S}$, we compute a pre-CST that has the same rich denotation as $\mathsf{CST\_Pre}(t, \mathbf{S})$: we set the value of a node n in the i-th layer to the following value:

$$val'(n) = max(d_{ti}, val(n) - c_{ti})$$

The max is defined as the conjunction between the guard and the result of the backward application of the action in t to x_i. We use the algorithm given in [36] (see Appendix A) to adjust the two conditions 5 and 6. Although the algorithm for ensuring condition 5 is exponential in the worst case, this method performs well in practice. The backward reachability algorithm based on CSTs is given in Figure 15. The algorithm computes the $\mathsf{CST\_Pre}$ operation on the variable $\mathbf{S}$ associated with the CST describing the set of minimal markings computed at every iteration. The CST stored in $\mathbf{S}$ may contain several redundant paths. The following proposition can help us to reduce its size.

function $\mathsf{Symbolic_{BS}}(\mathbf{S}_U : \mathrm{CST}, \mathbf{m}_0 : \mathit{marking}) : \mathit{boolean}$
  $\mathbf{S} \leftarrow \mathbf{S}_U$;
  $\mathbf{T} \leftarrow$ CST_Empty;
  while not(CST_Subsumption($\mathbf{T}, \mathbf{S}$)) **do**
    $\mathbf{T} \leftarrow$ CST_Union($\mathbf{T}, \mathbf{S}$);
    $\mathbf{R} \leftarrow$ CST_Empty;
    for each *transition* t **do**
      $\mathbf{N} \leftarrow$ CST_Pre($t, \mathbf{S}$);
      $\mathbf{R} \leftarrow$ CST_Union($\mathbf{N}, \mathbf{R}$);
    endfor
    $\mathbf{S} \leftarrow \mathbf{R}$;
  endwhile
  return CST_Sat($\mathbf{m}_0, \mathbf{T}$);
endfunction

Fig. 15. Basic backward reachability algorithm using CSTs.

Proposition 5. *Let* $\mathbf{m}$ *and* $\mathbf{m}'$ *two markings, and let* t *be a transition. If* $\mathbf{m} \preccurlyeq \mathbf{m}'$ *holds, then we have*

i) $\mathsf{CST\_Pre}(t, \mathit{cone}(\mathbf{m}')) \subseteq \mathsf{CST\_Pre}(t, \mathit{cone}(\mathbf{m}))$
ii) $\mathsf{CST\_Pre}^*(\mathit{cone}(\mathbf{m}')) \subseteq \mathsf{CST\_Pre}^*(\mathit{cone}(\mathbf{m}))$

¿From the previous property, all the paths of $\mathbf{S}$ subsumed by a path of $\mathbf{T}$ can always be removed during the exploration. Unfortunately, it has been shown in section 7 that this step cannot be done in polynomial time. However, we can still define efficient heuristics to remove some of the paths from $\mathsf{CST\_Pre}(t, \mathbf{S})$ that are subsumed by paths of $\mathbf{S}$.

For example, consider the transition t defined by the guard $x_2 \geq 1 \wedge x_4 \geq 1$ and by the action $x_2' = x_2 - 1$, $x_4' = x_4 - 1$, $x_1' = x_1 + 1$ and $x_5' = x_5 + 1$. If we apply t to the upward-closed set with the minimal marking $\langle 0, 2, 1, 4, 0 \rangle$, we obtain its minimal marking $\langle 0, 3, 1, 5, 0 \rangle$. Thus, the resulting marking does not yield new information and can be removed from $\mathbf{S}$ during the computation of $\mathsf{CST\_Pre}^*$. We will formalize this idea in the following proposition.

Proposition 6. *Given a marking* $\mathbf{m}$ *and a transition* t*, if for each* x_i *such that* $c_{ti} > 0$, $\mathbf{m}(x_i) \leq d_{ti}$ *holds, then it follows that* $\mathit{cone}(\mathsf{CST\_Pre}(t, \mathbf{m})) \subseteq \mathit{cone}(\mathbf{m})$

When we compute $\mathsf{CST\_Pre}(t, \mathbf{S})$, it is interesting to efficiently detect and remove some paths of $\mathbf{S}$ satisfying Proposition 6 before the computation of CST_Pre. The following proposition gives a way to detect that all the paths of $\mathsf{CST\_Pre}(t, \mathbf{S})$ are subsumed by paths of $\mathbf{S}$.

Proposition 7. *Let* t *be a transition of a Petri net* P *and* $\mathbf{S}$ *be a CST. If for all layers* i *such that* $c_{tj} > 0$ *and for all* $n \in N_i^{\mathbf{S}}$ *such that* $\mathit{val}^{\mathbf{S}}(n) \leq d_{ti}$*, then* $\mathit{cones}(\mathsf{CST\_Pre}(t, \mathbf{S})) \subseteq \mathit{cones}(\mathbf{S})$.

Checking condition of Proposition 7 can be done in linear time on CSTs and allow us to know if it is interesting to compute CST_Pre for a transition t on a CST $\mathbf{S}$ with a low cost.

When $\mathit{cones}(\mathsf{CST\_Pre}(t, \mathbf{S})) \not\subseteq \mathit{cones}(\mathbf{S})$ algorithm of Figure 16 can be used to reduce $\mathbf{S}$ by removing edges e

procedure PrunePre($\mathbf{S} : \mathrm{CST}, t : \mathit{transition}$)
  $\mathsf{ComputeSub}^{\downarrow}(\mathbf{S}, t)$;
  $\mathsf{ComputeSub}^{\uparrow}(\mathbf{S}, t)$;
  Remove all edges $e = \langle n, m \rangle$ *such that*
  $\mathit{Sub}^{\downarrow}(n) = \mathit{false} \wedge \mathit{Sub}^{\uparrow}(m) = \mathit{false}$;
endprocedure

Fig. 16. algorithm that removes paths generating redundant ones when applying CST_Pre.

procedure $\mathsf{ComputeSub}^{\downarrow}(\mathbf{S} : \mathrm{CST}, t : \mathit{transition})$
  for all $n \in N^{\mathbf{S}} : \mathit{Sub}^{\downarrow}(n) \leftarrow \mathit{false}$;
  for each $i : 1$ *to* k (*top ot bottom*) **do**
    for each *node* $n \in N_i^{\mathbf{S}}$ **do**
      if $\mathit{newval}(n) \neq \mathit{val}^{\mathbf{S}}(n) \wedge c_{tl} > 0$ **then**
        $\mathit{Sub}^{\downarrow}(n) \leftarrow \mathit{true}$;
      endif
      for each $n' \in \mathit{succ}^{\mathbf{S}}(n)$**do**
        if $\mathit{Sub}^{\downarrow}(n) = \mathit{true}$ **then**
          $\mathit{Sub}^{\downarrow}(n') \leftarrow \mathit{true}$;
        endif
      endfor
    endfor
  endfor
endprocedure

Fig. 17. algorithm of $\mathsf{ComputeSub}^{\downarrow}$.

such that all the paths passing by e define upward-closed set including their set of predecessors. $\mathsf{ComputeSub}^{\downarrow}$ is defined in Figure 17 (we assume that $\mathit{newval}(n)$ is the new value of n when computing the pre-CST corresponding to $\mathsf{CST\_Pre}(t, \mathbf{S})$). The function $\mathsf{ComputeSub}^{\uparrow}$ is defined from $\mathsf{ComputeSub}^{\downarrow}$ by traversing the CST from the last layer to the first one, replacing the relation $\mathit{succ}^{\mathbf{S}}$ by $\mathit{father}^{\mathbf{S}}$ and $\mathit{Sub}^{\downarrow}$ by $\mathit{Sub}^{\uparrow}$. The two following propositions holds.

Proposition 8. *Given a transition* t*, a CST* $\mathbf{S}$ *and the pre-CST* $\mathbf{S}'$ *obtained by applying* PrunePre *on* $\mathbf{S}$ *with* t *and then applying algorithm of Figure 34 that ensures condition 6, we have*

$$\mathsf{CST\_Pre}(\mathit{cones}(\mathbf{S})) = \mathsf{CST\_Pre}(\mathit{cones}(\mathbf{S}')).$$

Proposition 9. *The complexity of* PrunePre *is linear in the size of the input CST.*

Proof. Stems from the linear complexity of the breath-first search.

Another way to compute the CST_Pre operation is to follow the idea given in [36] by defining a sharing tree, called transition tree, describing the set of transitions. The value of the nodes of the transition tree is a couple where the first component describe a guard of transition (i.e. a constant c_{ti}), the second component describe an effect of a transition (i.e. a constant d_{ti}) and each path correspond to a transition. Figure 18 shows the transition tree for the transitions t_1 with guard $x_3 \geq 1$

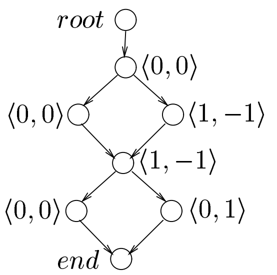

Fig. 18. Example of transition tree.

```
function Symbolic_BS(S_U : CST, m_0 : marking) : boolean
   S ← S_U;
   T ← CST_Empty;
   while not(CST_Subsumption(T, S)) do
      T ← CST_Union(T, S);
      R := CST_Empty;
      for each transition t do
         N ← CST_Pre(S, t);
         CST_Remove_Redundancies(N, R, T);
         CST_Heuristic_Minimization(N);
         R ← CST_Union(N, R);
      endfor
      S ← R;
   endwhile
   return CST_Sat(m_0, T);
endfunction
```

Fig. 19. Backward reachability algorithm using CSTs.

and effect $x_3' = x_3 - 1$, t_2 with guard $x_3 \geq 1$ and effect t $x_3' = x_3 - 1 \wedge x_4' = x_4 + 1$, t_3 with guard $x_2 \geq 1 \wedge x_3 \geq 1$ and effect $x_2' = x_2 - 1 \wedge x_3' = x3 - 1$ and transition t_4 with guard $x_2 \geq 1 \wedge x_3 \geq 1$ and effect $x_2' = x_2 - 1 \wedge x_3' = x3 - 1 \wedge x_4' = x_4 + 1$. By computing the operation corresponding to the product of automata between the transition tree and a CST **S** ,we compute CST_Pre(**S**) in one step (i.e. according to all the transitions). For a node $\langle n, m\rangle$ of the product where n is a node of the starting CST and m is a node of the transition tree, its value is computed as before according to the value of n and the guard and effect defined by m. This product can be done in polynomial time but the resulting pre-CST could violate condition 5-6. The algorithm of Figure 33 and Figure 34 must be then applied to obtain a CST.

Note that to compute efficiently the CST_Pre operation, we can use an algorithm similar to PrunePre.

9.7 The CST-based Backward Search Engine

We are ready now to formulate our CST-backward reachability algorithm. The algorithm in pseudo code is given in Figure 19. The algorithm works as follows. First of all, we store intermediate results in two variables. The variable **S** stores the current *frontier* of the breadth-first search performed by the algorithm. The variable **T** stores the set of visited generators.

Before entering the main loop, we test subsumption between **S** and **T**. For this purpose, the following heuristics seems to work well in practice.

1. We first compute the forward and backward simulation relations between the nodes of **S** and the nodes of **T**.
2. If the root of **S** is forward simulated by the root of **T** or if the end node of **S** is backward simulated by the end node of **T**, then we know that all the generators of **S** are subsumed by some generators of **T** (see [14]), thus the fixpoint is reached.
3. If the test fails, we perform a depth-first, top-down visit of the CST **S** in order to compare its tuples with those of **T**. During the depth-first visit, we use however the information previously computed via the forward simulation as follows. Each time we reach a node n that is forward simulated by a node of **T**, we stop the exploration: all the elements in the subtree rooted at n will be subsumed by elements of **T**.

In the main loop, we compute the new frontier **N** transition-by-transition via the symbolic operator CST_Pre. As a further heuristics to keep the size of **N** small, after computing CST_Pre, we apply simulation-based heuristics to remove *redundant* generators from the frontier. The function CST_Remove_Redundancies(**N**, **R**, **T**) uses simulations relation between nodes of **N** and nodes of **N** and **T**. The function CST_Heuristic_Minimization(**N**) uses simulations relation of nodes of **N**.

9.8 Preliminary Experiments

Let us go back to the example of the Manufacturing System discussed in Section 9.3. The use of CSTs as assertional language to represent intermediate results dramatically speeds up the analysis of the verification problem proposed in [12]. The computation based on the CST-search engine of Figure 19 terminates after 39 seconds on a Sun Ultra Sparc (see Figure 20). In a first experiment we have not removed the redundancies from $\mathbf{S}_{pre^i}$, whereas in a second experiment we have applied the reduction based on the forward simulation (every 5 steps) (see Fig 20). Simulation-based reduction (every 5 steps) allows us to reduce the set of generators of a factor of ten (note: removing *all* redundancies yields 450 elements).

In the next section we will show how to further speed up backward analysis by considering heuristics inspired to the structural theory of Petri Nets.

10 Structural Heuristics

In the backward reachability approach every place of a Petri Net is considered as *unbounded*. In many practical

Example	NV	NR	NS	LR	FSR	ET	NE	Nodes	Ratio
Man. Sys.	13	6	24	no	__	39s	7563	4420	0.04
Man. Sys.	13	6	24	yes	5	68s	727	1772	0.18

Fig. 20. Flags: NV=No. Variable; NR=No. Rules; ET=Execution Time; LR=Use of local reduction; FSR=Frequency in the use of sim. reduction (__=not used); NS=No Steps; NE=No. of Elem. of the result sharing tree; Ratio=Nodes/(NV*NE).

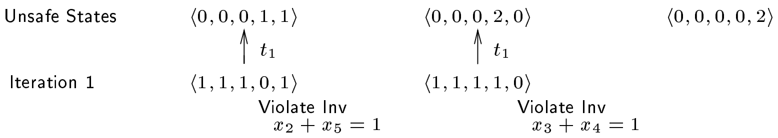

Fig. 21. Backward search space pruned using structural invariants.

cases, however, some places of the net might be bounded for any possible value of the parameters in the initial configuration. The *Structural Theory* of Petri Nets [34] can help us to distinguish between bounded and unbounded places and, thus, to restrict the search space we have to explore backwards. Let us recall the main results of this theory.

Let N be a Petri Net with n places, m transitions, and *token flow matrix* $\mathbf{C}$. We recall that $\mathbf{C}$ describes how tokens are reallocated by transitions: rows correspond to places, columns to transitions, and an element c_{ij} to the difference between the number of tokens added and removed for place i by the j-th transition. In the following, we will use $\cdot$ to denote the vector product

$$\mathbf{a}^T \cdot \mathbf{b} = a_1 b_1 + \dots a_n b_n$$

where $\mathbf{a}^T$ indicates the transpose of vector $\mathbf{a}$.

Place invariants [34] are among the possible information we can compute via a static analysis of N. A place invariant (also called P-semiflow) is a vector $\mathbf{p} = \langle p_1, \dots, p_n \rangle$ with non-negative components that is a solution of the linear system

$$\mathbf{x}^T \ \cdot \ \mathbf{C} \ = \ 0, \ \mathbf{x} \geq \mathbf{0},$$

where $\mathbf{x}$ is a vector of variables of dimension n. Given an initial marking $\mathbf{m_0}$, and a place invariant $\mathbf{p}$, the set

$$\mathcal{O}(\mathbf{m_0}, \mathbf{p}) = \{\mathbf{m} \mid \mathbf{p}^T \cdot \mathbf{m} = \mathbf{p}^T \cdot \mathbf{m_0}\}$$

over-approximates the reachability set of the Petri Net. This property follows from the definition of place invariant, and from the *state equation*

$$\mathbf{m} = \mathbf{m_0} + \mathbf{C} \cdot \sigma$$

that characterizes a generic marking $\mathbf{m}$ reachable from $\mathbf{m_0}$ via the sequence of transitions represented by the *firing vector* σ (see [34]): if $\sigma = \langle c_1, \dots, c_m \rangle$ then the j-th transition has been fired c_j times to reach $\mathbf{m}$ from $\mathbf{m_0}$. As a consequence, the equation

$$\mathbf{p}^T \cdot \mathbf{x} = \mathbf{p}^T \cdot \mathbf{m_0}$$

for some place invariant $\mathbf{p}$ gives us a *structural invariant* we can use to analyze the net.

Let us consider the example of Figure 11. The state equation is given as follows

$$\begin{pmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \\ m_5 \end{pmatrix} = \begin{pmatrix} K \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} -1 & 1 & 1 & -1 \\ 0 & 1 & 0 & -1 \\ -1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & -1 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{pmatrix}$$

where $m_1, \dots, m_5$ and $c_1, \dots, c_5$ are required to be greater or equal than zero. Place invariants can be computed then by solving the following linear system

$$\begin{pmatrix} x_1 \ x_2 \ x_3 \ x_4 \ x_5 \end{pmatrix} \cdot \begin{pmatrix} -1 & 1 & 1 & -1 \\ 0 & 1 & 0 & -1 \\ -1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & -1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 \ 0 \ 0 \ 0 \end{pmatrix}$$

where $y_1, \dots, y_5$ are required to be greater or equal than zero. After some simplifications, we get the following linear system

$$\begin{gathered} x_1 + x_2 = x_5 \\ x_1 + x_3 = x_4 \\ x_1, x_2, x_3, x_4, x_5 \geq 0 \end{gathered}$$

This gives us, for instance, the following *place invariants*: $\mathbf{p_1} = \langle 0, 1, 0, 0, 1 \rangle$, $\mathbf{p_2} = \langle 0, 0, 1, 1, 0 \rangle$, and $\mathbf{p_3} = \langle 1, 0, 0, 1, 1 \rangle$. Actually, these place invariants represent a basis to generate (via linear combinations) all other place invariants. Algorithms to compute generators are mentioned, e.g., in [34]. When applied to the state equation and to the parametric initial marking, we obtain then the following structural invariants:

$$\begin{gathered} x_2 + x_5 = 1 \\ x_3 + x_4 = 1 \\ x_1 + x_4 + x_5 = K. \end{gathered}$$

Notice that the first two invariants, namely $x_2 + x_5 = 1$ and $x_3 + x_4 = 1$, imply that the places x_2, x_3, x_4 and x_5 always contain at most one token, i.e., they imply

$x_i \leq 1$ for $i : 2, \ldots, 5$. Unfortunately, these information are not sufficient to prove our mutual exclusion property expressed via the constraint $x_4 + x_5 \leq 1$: a configuration in which both x_4 and x_5 have one token satisfies the structural invariant $x_i \leq 1$ for $i : 2, \ldots, 5$ but not the mutual exclusion property. The reason of the weakness of the invariants is the fact that they do not take into account the guards of the transitions, an information that is not represented in the matrix $\mathbf{C}$. Despite of this problem, we can still try to exploit the information coming from the first two invariants.

A possible way to use the structural analysis would be to make what is usually called a *program specialization*, i.e., we could replace the subnet involving places linked by structural invariants (e.g. p_2, p_5 for $x_2 + x_5 = 1$) with a control part (a finite-state net). This way however, we run the risk of obtaining nets of exponential size. As an alternative, we propose to use the structural invariants directly within our CSTs-based search scheme as heuristics to *prune the search space*.

10.1 Pruning the Backward Search Space

Let U be an upward-closed set of markings denoting *unsafe states*, and let $U' = U \cap \mathcal{O}(\mathbf{m_0}, \mathbf{p})$ for some place invariant $\mathbf{p}$. We first note that if $U' = \emptyset$, then we can immediately infer that the net is *safe* with respect to the violations represented via U. As mentioned in the previous section, invariants are often not sufficient to directly verify the property, but they can be used to prune the backward search space. Let us consider again our example and the backward reachability graph of Figure 21. After the first iteration, we obtain the generators $\langle 1,1,1,0,1\rangle$ and $\langle 1,1,1,1,0\rangle$. They are not subsumed by pre-existing generators. The first generator defines a set of markings that has no intersection with the set of markings defined by the invariant $x_2 + x_5 = 1$, while the second generator defines a set of markings that have no intersection with the set of markings defined by the invariant $x_3 + x_4 = 1$. So we know that no markings defined by those two generators will be reached from any instance of the parametric initial marking (the invariants do not depend on the value of K). In fact, the set of markings satisfying the invariant over-approximates the set of reachable markings. As a consequence, we can stop the backward search after the first iteration.

Let us now study how we can incorporate this idea in our CST-based backward search. We first note that, in general, the intersection between an upward-closed set U and the set of markings $\mathcal{O}(\mathbf{m_0}, \mathbf{p})$ induced by a place invariants $\mathbf{p}$ is not upward-closed. Thus, even if we work using a symbolic representation of invariants using linear constraints (e.g. $x_2 + x_5 = 1 \wedge x_3 + x_4 = 1$), we cannot conjunct the invariant with the intermediate results of the algorithm for pre^* and represent the result with a CST. For instance, the constraint $x_2 + x_5 = 1 \wedge$ $x_2 \geq 1 x_5 \geq 0$ is equivalent to $x_2 = 1 \wedge x_5 = 0$, whose denotation is not upward-closed.

What we can do, however, is to remove, from the CSTs computed after each step of the symbolic reachability algorithm, *all the generators* that define sets of markings that have no intersection with the set $\mathcal{O}(\mathbf{m_0}, \mathbf{p})$. In fact, we have the following property.

```
function Symbolic_BS(S_U : CST, m_0 : marking) : boolean
   S ← S_U;
   T ← CST_Empty;
   while not(CST_Subsumption(T, S)) do
      T ← CST_Union(T, S);
      R ← CST_Empty;
      for each transition t do
         N ← CST_Pre(S, t);
         CST_Structural_Reduction(N);
         CST_Remove_Redundancies(N, R, T);
         CST_Heuristic_Minimization(N);
         R ← CST_Union(N, R);
      endfor
      S ← R;
   endwhile
   return CST_Sat(m_0, T);
endfunction
```

Fig. 22. Backward reachability algorithm using CSTs enriched with pruning.

Theorem 17. *Given a Petri Net N with initial parametric marking $\mathbf{m_0}$, a place invariant $\mathbf{p}$, and an upward-closed set of markings U represented by a CST $\mathbf{S}$, suppose $cones(\mathbf{m}) \cap \mathcal{O}(\mathbf{m_0}, \mathbf{p}) = \emptyset$ for some $\mathbf{m} \in elem(\mathbf{S})$. Furthermore, let $\mathbf{S}'$ be the CST such that $elem(\mathbf{S}') = elem(\mathbf{S}) \setminus \{\mathbf{m}\}$, and $\mathbf{m_0'}$ be any instance of $\mathbf{m_0}$. Then,*

$$\textit{If } \mathbf{m_0'} \in pre^*(cones(\mathbf{S})) \textit{ then } \mathbf{m_0'} \in pre^*(cones(\mathbf{S}')).$$

Proof. Per absurdum, suppose there exists an instance $\mathbf{m_0}'$ of $\mathbf{m}_0$ such that $\mathbf{m_0'} \in pre^*(cones(\mathbf{S}))$ and $\mathbf{m_0'} \notin pre^*(cones(\mathbf{S}'))$. This implies that from $\mathbf{m_0'}$ we can reach a marking $\mathbf{m}'$ generated by $\mathbf{m}$. But this contradict the hypothesis that $\mathbf{p}$ is a place invariant of N with $cones(\mathbf{m})$ $\cap \mathcal{O}(\mathbf{m_0}, \mathbf{p}) = \emptyset$.

Thus, during the backward search we can prune the search space by removing *all* generators $\mathbf{m}$, such that $cone(\mathbf{m})$ has an empty intersection with the set of markings defined by a structural invariant. We call such generators *useless*. As shown in the following theorem, this process can be done in polynomial time in the size of the CSTs.

Theorem 18. *Given a Petri Net N with initial marking $\mathbf{m_0}$, and a place invariant $\mathbf{p}$, such that $M = \mathbf{p}^T \cdot \mathbf{m_0}$ is an integer value (no parameters occur in M). Then, there exists an algorithm to compute the CST $\mathbf{S}'$ such that $\mathbf{m} \in elem(\mathbf{S}')$ if and only if $\mathbf{m} \in elem(\mathbf{S})$ and $cone(\mathbf{m}) \cap \mathcal{O}(\mathbf{m_0}, \mathbf{p}) \neq \emptyset$, with complexity polynomial in the size of $\mathbf{S}$, $\mathbf{p}$, and in the value M.*

Proof. Let $\mathbf{p} = \langle p_1, \ldots, p_k \rangle$ and $M = \mathbf{p}^T \cdot \mathbf{m_0}$.

The algorithm works as follows. We first constructs (in quadratic time) a pre-sharing tree $\mathbf{T}$ such that $\mathbf{m} \in elem(\mathbf{T})$ if and only if $\mathbf{m} \in elem(\mathbf{S})$ and $cone(\mathbf{m}) \cap \mathcal{O}(\mathbf{m_0}, \mathbf{p}) \neq \emptyset$.

$\mathbf{T}$ is defined as follows. Its set of nodes $N^{\mathbf{T}}$ consists of $root^{\mathbf{T}}$, $end^{\mathbf{T}}$, and the nodes of the i-th layer (in $N_i^{\mathbf{T}}$) are the pairs $\langle s, v \rangle$ composed by a node $s \in N_i^{\mathbf{S}}$, and a value $v \in \{0, \ldots, M\}$. We set $val(\langle s, v \rangle) = val(s)$. The successor relation $succ^{\mathbf{T}}$ is defined as (recall that $\mathbf{p} = \langle p_1, \ldots, p_k \rangle$):

- $succ^{\mathbf{T}}(root^{\mathbf{T}})$ contains all nodes $\langle s, v \rangle$ such that $s \in succ^{\mathbf{S}}(root^{\mathbf{S}})$ and $v = p_1 \cdot val(s)$;
- for all the nodes $\langle s, v \rangle$ of the ℓ-th layer ($\ell < k$) $succ^{\mathbf{T}}(\langle s, v \rangle) = \{ \langle s', v' \rangle$ such that $s' \in succ^{\mathbf{S}}(s)$ and $v' = v + p_{\ell+1} \cdot val(s') \}$; note that if $p_i = 0$ then $\langle s', 0 \rangle \in succ^{\mathbf{T}}(\langle s, v \rangle)$ if and only if $s' \in succ^{\mathbf{S}}(s)$.
- for all nodes $\langle s, v \rangle$ of the k-th layer $succ^{\mathbf{T}}(\langle s, v \rangle) = \{ end^{\mathbf{T}} \}$.

The resulting pre-sharing tree has the following property: If $s' \in succ^{\mathbf{S}}(s)$ then for each value $v \in \{0, \ldots, M\}$, there is *at most one* v' such that $\langle s, v' \rangle \in succ^{\mathbf{T}}(\langle s, v \rangle)$. Furthermore, $\mathbf{t} \in elem(\mathbf{T})$ if and only if $\mathbf{t} \in elem(\mathbf{S})$ and $\mathbf{p}^T \cdot \mathbf{t} = \mathbf{p}^T \cdot \mathbf{m_0}$. The construction of $\mathbf{T}$ is done in time *polynomial* in the size of $\mathbf{S}$, M and $\mathbf{p}$. Finally, to turn $\mathbf{T}$ into a CST we have to enforce the maximal sharing of suffixes in $\mathbf{T}$. This can be also done in *polynomial* time using the algorithm proposed in Figure 34. □

As we only need $\lceil log_2(M) \rceil$ bits to encode a value M, the algorithm presented in the proof of Theorem 18 is only a *pseudo* polynomial algorithm. In the next section, we present a polynomial heuristics to remove useless generators.

10.2 Approximation of the Structural Heuristic

In the previous section we presented a pseudo polynomial algorithm to remove all the *useless* paths of a CST according to a structural invariant. We present now an approximated algorithm that can be used as an heuristics to remove *useless* paths. The new algorithm is more efficient in practical cases. To describe the heuristics, we need the following definitions. Let $\mathbf{S}$ be a CST , and let $e = (\mathbf{v}, \mathbf{w})$ be an edge of $\mathbf{S}$ connecting nodes of two adjacent layers. We define $elem_e(\mathbf{S})$ as the set of tuples from $elem(\mathbf{S})$ denoted by paths of $\mathbf{S}$ passing through e. Formally, $\mathbf{m} = \langle val(v_1), \ldots, val(v_k) \rangle \in elem_e(\mathbf{S})$ if and only if there exists a path $\langle root^{\mathbf{S}}, v_1, \ldots, \mathbf{v}, \mathbf{w}, \ldots, v_k, end^{\mathbf{S}} \rangle$ in $\mathbf{S}$. Consider now a *structural invariant*, say $\mathcal{I}$, having the form $\mathbf{p}^T \cdot \mathbf{x} = \mathbf{p}^T \cdot \mathbf{m_0}$.

Our heuristics works by removing an edge e of $\mathbf{S}$, whenever we can prove that the markings in $elem_e(\mathbf{S})$ denote cones that do not intersect with the structural invariant $\mathcal{I}$. To check this condition on the edge e connecting a node of layer i and a node of layer $i+1$, we first compute the two values $min_{\prec}(e)$ and $min_{\succ}(e)$ defined as follows: $min_{\prec}(e)$ is the minimal value of *prefixes* $\langle m_1, \ldots, m_i \rangle$ of tuples in $elem_e(\mathbf{S})$ evaluated on the function $\mathbf{x} \rightsquigarrow \mathbf{p}^T \cdot \mathbf{x}$; symmetrically, $min_{\succ}(e)$ is computed for *suffixes* $\langle m_{i+1}, \ldots, m_k \rangle$. Specifically, we define

$$min_{\prec}(e) = min \ \{ \ \mathbf{p}^T \cdot \langle m_1, \ldots, m_i, 0, \ldots, 0 \rangle \mid \langle m_1, \ldots, m_i, \ldots, m_k \rangle \in elem_e(\mathbf{S}) \}$$

$$min_{\succ}(e) = min \ \{ \ \mathbf{p}^T \cdot \langle 0, \ldots, 0, m_{i+1}, \ldots, m_k \rangle \mid \langle m_1, \ldots, m_{i+1}, \ldots, m_k \rangle \in elem_e(\mathbf{S}) \}.$$

The following two properties characterize our heuristics.

```
procedure min≻(S : CST, p : structural invariant)
    for each e = ⟨n, n'⟩ such that n' ∈ succ^S(n) do
        min≻(e) ← 0;
    endfor
    minprec(end^S) ← 0;
    for each i : k to 1 (bottom−up) do
        for each node n ∈ N_i^S do
            minprec(n) ← ∞;
            for each e = ⟨n, n'⟩ s.t. n' ∈ succ^S(n) do
                min≻(e) ← minprec(n');
                minprec(n) ← min(minprec(n), minprec(n'));
            endfor
            minprec(n) ← minprec(n) + p_i · val(n);
        endfor
    endfor
    for each e = ⟨root^S, n⟩ such that n ∈ succ^S(root^S) do
        min≻(e) ← minprec(n);
    endfor
endprocedure
```

Fig. 24. algorithm computing $min_{\succ}$ for each edges of a CST $\mathbf{S}$.

Theorem 19. *Given an initial marking* $\mathbf{m_0}$*, a CST* $\mathbf{S}$*, a structural property* $\mathbf{p}^T \cdot \mathbf{x} = \mathbf{p}^T \cdot \mathbf{m_0}$ *with* $\mathbf{p}^T \cdot \mathbf{m_0} \in \mathbb{N}$*, and an edge* e *of* $\mathbf{S}$*, if* $min_{\prec}(e) + min_{\succ}(e) > \mathbf{p}^T \cdot \mathbf{m_0}$*, then* $cone(\mathbf{m}) \cap \mathcal{O}(\mathbf{m_0}, \mathbf{p}) = \emptyset$ *for any* $\mathbf{m} \in elem_e(\mathbf{S})$.

Theorem 20. *Given a CST* $\mathbf{S}$*, and a structural invariant* $\mathbf{p}^T.\mathbf{x} = \mathbf{p}^T.\mathbf{m_0}$ *such that* $\mathbf{p}^T.\mathbf{m_0} \in \mathbb{N}$*, the computation of* $min_{\prec}$ *and* $min_{\succ}$ *for each* $e = \langle n, n' \rangle$ $(n, n' \in N^{\mathbf{S}})$ *such that* $n' \in succ(n)$ *can be done in time* $\mathcal{O}(|\mathbf{S}|)$.

Proof. The algorithm of Figure 24 computes $min_{\succ}$ on a CST $\mathbf{S}$ for all $e = \langle n, n' \rangle$ such that $n' \in succ^{\mathbf{S}}(n)$. The algorithm is linear in the size of the input CST. In fact, it amounts to a breadth-first search. The algorithm that computes $min_{\prec}$ can be defined starting from the algorithm computing $min_{\succ}$ by replacing the relation $succ^{\mathbf{S}}$ by the relation $father^{\mathbf{S}}$, $min_{\succ}$ by $min_{\prec}$ and traversing the CST from the first layer to the last one.

Based on this results, the algorithm that approximates the removing of the *useless* paths is presented in Figure 25. This algorithm has a running time linear in the size of the input CST.

As an example of application of the structural heuris-

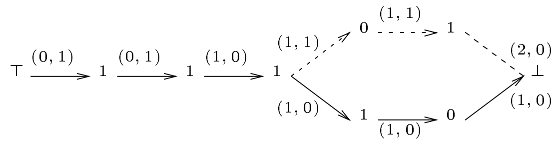

Fig. 23. Application of Heuristics for Structural Invariants to a CST.

procedure ApproxSH($\mathbf{S}$: CST, $\mathbf{p}$: *structural invariant*, $\mathbf{m}_0$: *marking*)
  $\mathsf{min}_{\succ}(\mathbf{S}, \mathbf{p})$;
  $\mathsf{min}_{\prec}(\mathbf{S}, \mathbf{p})$;
  Remove $e = \langle n, n' \rangle$ *such that*
  $\min_{\succ}(e) + min_{\prec}(e) > \mathbf{p}^T \cdot \mathbf{m_0}$;
endprocedure

Fig. 25. Algorithm that approximates the removing of useless paths.

tics, consider the CST of Figure 23. The CST $\mathbf{S}$ contains the elements obtained at iteration 1, the pairs of values on the arcs are the values $min_{\prec}(e)$ and $min_{\succ}(e)$ for the place invariant $x_2 + x_5 = 1$, the dashed edges can be removed and thus the useless element $\langle 1, 1, 1, 0, 1 \rangle$ is removed from the CST. Note that if we use the invariant $x_3 + x_4 = 1$ then the last element can also be eliminated. The heuristics simply traverses a CST layer by layer, removing all edges that satisfy the hypothesis of Theorem 19.

Based on the previous ideas, the CST-based symbolic reachability algorithm of Figure 19 can be enriched as shown in Figure 22. Given a set of place invariants computed a priori via a static analysis of a Petri Net, the procedure CST_Structural_Reduction implements the *reduction algorithm* based on structural heuristics of Theorem 18 to removes all *useless* generators from the input CST. To complete the scenario, we need to compute *automatically* the structural invariants. This can be done using specialized libraries to compute systems of generators for place invariants like the one available with GreatSPN [10], or PEP [24].

11 Experimental Evaluation

In the previous sections, we have presented several algorithms for the backward analysis of Petri nets. We have also proposed a heuristic based on structural invariants to prune the backward search space. In this section, we evaluate the practical behavior of the different algorithms and the structural heuristic on several examples of infinite-state Petri nets. More precisely, we have considered *mutual exclusion* properties for the parameterized, concurrent and production systems like the Mesh 2x2 of [2] (Figure 130, p. 256), its extension to the 3x2 case, and for an extension of the Readers-Writers example given in [33] in which we use several large buffers (e.g. with 45 slots). Furthermore, we have considered *semi-liveness* and *coverability* problems for the PNCSA communication protocol analyzed in [3,20]. We also compare our work with alternative approaches.

In Section 11.1, we report on results obtained with the exact symbolic algorithms that we propose and an intelligent implementation of the standard enumerative algorithm. In Section 11.2, we compare the exact symbolic algorithm with a version where the simulation based heuristics are used. In Section 11.3, we show experimental evidences of the benefits of using structural invariants to guide the backward search. Finally, in Section 11.4, we compare our prototype with other tools for the analysis of infinite-state and finite-state Petri nets. We present some links that exists with other symbolic structures that can be used to represent infinite set of states and in particular upward-closed sets.

11.1 Enumerative versus Symbolic

In the previous sections we have shown that we are able to explore the backward reachable states of an infinite-state Petri net starting from an upward-closed set of final markings without enumerating the minimal points of the successive upward-closed sets computed in the fixpoint computation. We have given an algorithm that manipulates directly the graph structure of the CSTs representing compactly those minimal points.

We have also seen that some problems on CSTs, related to the symbolic manipulation of minimal elements of an upward-closed set, have a relatively high theoretical complexity (coNP-completeness for example). We show in this subsection that despite the theoretical complexities of the symbolic operations required for the backward search, the CSTs based algorithm outperforms a good implementation of the enumerative algorithm. Note that the term *enumerative* can be misleading: only the finite number of minimal elements of the infinite upward-closed sets are enumerated, not all the elements of those sets.

Before presenting our results, we need to be more specific about the enumerative version of the algorithm that we have used. A high level version of the algorithm that we have used is presented in figure 27.

The sets $\mathbf{R}$, $\mathbf{F}$, and $\mathbf{T}$ are maintained as CSTs. We do that in order to isolate as much as possible the effect of enumerating the elements of the frontier instead of manipulating these sets of minimal elements symbolically. Note that this way, the union operation and the tests $\neg\exists s'' \in \mathbf{T} \cup \mathbf{S} \cup \mathbf{R}$ s.t. $s'' \leq s'$ and $\exists s \in elem(\mathbf{T})$ s.t.

Size of the Petri Net
P=Number of places;
T = Number of transitions.
Verification problem (VP) (only the type of property)
ME=Mutual exclusion property;
C=Covering for a random marking;
SL=Semi-liveness.
Use of heuristics
I=Invariant-based reductions (structural heuristic rule);
S=Simulation-based reductions.
Statistics: execution
EX=Execution time (in seconds) on an AMD Athlon 900 Mhz;
NI=N. of iterations before termination (with *=before stopping the execution).
Quality of analysis
R=An initial state has been reached.
Statistics: use of memory
MaxE(N)=N. of *elements* (*nodes*) of the biggest CST associated with **S** of Figure 19;
NE(N)=Number of *elements* (*nodes*) of the CST for the fixpoint;
Ratio of memory saving (using CSTs)
RM=MAX-N/(MAX-E $\times$ P) in pct.;
RN=NN/(NE $\times$ P) in pct.

Fig. 26. Parameters of the experimental evaluation.

function $\mathsf{Enumerative}_{\mathsf{BS}}(\mathbf{S} : \mathrm{CST}, \mathbf{m}_0 : marking)$
$: boolean$
$\mathbf{F} \leftarrow \mathbf{S}$; *Frontier*
$\mathbf{T} \leftarrow \emptyset$; *Already visited*
while $\mathbf{F} \neq \emptyset$ **do**
$\mathbf{T} \leftarrow union(\mathbf{T}, \mathbf{F})$;
$\mathbf{R} \leftarrow \emptyset$;
for each $\mathbf{s} \in elem(\mathbf{F})$ **do**
for each *transition* t **do**
$\mathbf{s}' \leftarrow Pre_{min}(t, \mathbf{s})$;
if $\nexists \mathbf{s}'' \in elem(\mathbf{T}) \cup elem(\mathbf{S}) \cup elem(\mathbf{R})$
such that $\mathbf{s}'' \preccurlyeq \mathbf{s}'$ **then**
$elem(\mathbf{R}) \leftarrow elem(\mathbf{R}) \cup \{\mathbf{s}'\}$;
endif
endfor
endfor
$\mathbf{F} \leftarrow \mathbf{R}$;
endwhile
if $\exists \mathbf{s} \in elem(\mathbf{T}) : \mathbf{s} \preccurlyeq \mathbf{m_0}$ **then**
return *true*;
else
return *false*;
endif
endfunction

Fig. 27. The enumerative algorithm for the backward search.

$s \preccurlyeq \mathbf{m_0}$ can be done efficiently using the union algorithm on sharing trees described in [37] and the algorithm given in the proof of Theorem 4. So, the enumerative search is not naive and the comparison with the symbolic one is fair.

For the smaller examples the enumerative search has execution times that are from two to three times larger than the symbolic search. Note that the largest frontier as well as the fixpoint has the same size in the two methods. This is not surprising as the two methods maintain the intermediate results as CSTs. For the largest examples, the symbolic method works far better. In fact, for the second property of the PNCSA example, the symbolic method is able to compute the 38 iterations needed to reach the fixpoint, whereas the enumerative methods is only able to compute 11 iterations. The advantage of working symbolically is also apparent in the R/W example where the symbolic method is able to compute in 4 hours 19 iterations of the fixpoint computation and treat a frontier of up to 200000 elements when the enumerative algorithm is only able to compute 11 iterations in the same time and seems to be limited to treat frontiers of less than 10000 minimal elements in reasonable time. So working symbolically seems to be the right way to go. In the next two subsections, we will examine the practical behavior of two heuristics that can potentially further improve the efficiency of the symbolic algorithm.

11.2 Simulation-based vs Exact Minimization

As discussed in the previous sections, several basic operations of the *exact* backward symbolic search can not be done in polynomial time (unless P=NP). However we have proposed exact algorithms that are symbolic and only exponential in the worst-case. The exponential behavior can only occur when the underlying automaton structure of the CST is made deterministic. Experiments should determine if this pathological behavior often occurs in practice. Because of this hardness result, we have proposed polynomial approximation methods based on the notion of simulation. When using the sim-

Case-study	P	T	VP	EX(sec)	NI	MAX-E	MAX-N	RM	NE	NN	RN	Mem
Mesh2x2$_{k\geq 1}$	32	32	ME	1s	15	53	323	19.2	427	344	2.5	1676 kb
Mesh3x2$_{k\geq 1}$	52	54	ME	18.84s	21	332	1253	7.6	2224	987	0.8	1984 kb
PNCSA	31	36	SL	35.28	17	540	1967	11.7	89	176	6.7	1960 kb
PNCSA	31	36	C	>4h	11∗	5042	11323	7.2	–	–	–	–
R/W$_{k\geq 1}$	24	22	ME	>4h	12∗	15449	5996	1.6	–	–	–	–

Fig. 28. Experiments using the enumerative algorithm.

Case-study	P	T	VP	EX(sec)	NI	MAX-E	MAX-N	RM	NE	NN	RN	Mem
Mesh2x2$_{K\geq 1}$	32	32	ME	0.8	15	53	323	19	427	344	2.5	1932 kb
Mesh3x2$_{K\geq 1}$	52	54	ME	6.5	21	332	1253	7.3	2224	987	0.9	3492 kb
PNCSA	31	36	SL	17.65	17	540	2100	12.5	89	176	6.7	3748kb
PNCSA	31	36	C	3h50min	38	8545	17468	6.6	132	368	9	42772kb
R/W$_{K\geq 1}$	24	22	ME	>4h	19*	205102	27112	0.5	–	–	–	–

Fig. 29. Experiments using the symbolic algorithm keeping the CSTs in normal form.

ulation heuristics, the structure remains deterministic and no determinization is necessary. But the simulation based algorithms are not exact and even if in the fixpoint the sets computed at each iterations are the same as in the exact method, the CSTs that represent those sets may contain some redundancy, i.e. the CSTs may contain elements that are not minimal ones. We compare here the practical behavior of the two approaches: the exact method never generate redundant element but use algorithms with exponential worst-case behavior while the simulation based heuristic use polynomial time algorithms (with the exception of the fixpoint detection algorithm) but may generate some redundancy in the CSTs.

In the examples that we have tested so far, the exponential worst-case behavior of the exact algorithm does not occur and the performances of the exact algorithm are better than the performances of the simulation based algorithm. In the small examples, we can see that the heuristic does not generate too much redundancy, in fact, CSTs of the maximal frontiers as well as the ones of the fixpoints only contain a small number of redundant elements (difference between the number of elements in the simulation based search and the exact search in the tables of Figure 29 and Figure 30). The simulation based algorithm should not be judged useless, however. In fact, the exact algorithm may have a pathological behavior on some problems and it could be worth to test the simulation based algorithm in those cases.

11.3 Practical Evaluation of the Structural Heuristics.

In section 10, we have proposed to use structural invariants to prune the backward search space. We show here that this heuristic is very effective in practice and allow us to verify systems much more efficiently and more importantly systems that are out of reach of the method without this heuristic.

In all the examples, the use of the structural heuristic improves dramatically the performance of the exact symbolic backward search. The covering property for the PNCSA is solved using the heuristic within $43.09s$ instead of $3h50m$ (that is around 320 times faster). The gain when verifying the R/W example is even clearer. The exact symbolic algorithm without the invariant heuristic is only able to compute the first 19 iterations of the fixpoint computation in 4 hours when the 118 iterations necessary to reach the fixpoint are computed in less than 10 minutes when the structural heuristic is used.

11.4 Comparison with HyTech

HyTech has been developed for the forward and backward analysis of linear hybrid automata. HyTech uses polyhedra in the reals to represent possible valuations for the continuous variables of the hybrid automata. The linear operations involved in the backward analysis of Petri nets can equivalently be evaluated in the positive reals or the natural numbers. Furthermore, cones are special case of (unbounded) polyhedra. As a consequence, we can use HyTech to verify infinite-state Petri nets. We have run HyTech on the examples and include invariants to cut the search space. The results are given in the table of Figure 32.

The performances of HyTech on this class of infinite-state systems are far worst than the performance of our prototype. The comparison allow us to conclude that a general data structure like polyhedra used in HyTech can not compete with our data structure specialized for manipulating complex upward-closed sets. The main problem with the polyhedra library seems the memory consumption : every example but the simpler leads to a memory overflow. More than $500Mb$ are used by the algorithm based on polyhedra after a few iterations while our algorithm use less than $35Mb$ to solve completely the largest example.

Case-study	P	T	VP	EX(sec)	NI	MAX-E	MAX-N	RM	NE	NN	RN	Mem
Mesh2x2$_{k\geq 1}$	32	32	ME	1s	15	54	319	18.4	432	338	2.4	2148 kb
Mesh3x2$_{k\geq 1}$	52	54	ME	12.94s	21	334	1198	6.9	2444	1088	0.9	4112 kb
PNCSA	31	36	SL	30.05s	17	548	1972	11.6	89	176	6.4	5092 kb
PNCSA	31	36	C	> 14400	15∗	23700	29120	4	–	–	–	–
R/W$_{k\geq 1}$	24	22	ME	> 14400	16∗	129086	24037	0.8	–	–	–	–

Fig. 30. Experiments using the symbolic algorithm using the simulation relations to approximate the normal form of the CSTs.

Case-study	P	T	VP	EX(sec)	NI	MAX-E	MAX-N	RM	NE	NN	RN	Mem
Mesh2x2$_{K\geq 1}$	32	32	ME	0.8	15	53	323	19	424	340	2.5	2580Kb
Mesh3x2$_{K\geq 1}$	52	54	ME	6.5	21	326	1231	7.3	2209	985	1	4996Kb
PNCSA	31	36	SL	2.86	17	85	510	19.3	89	176	6.4	2596kb
PNCSA	31	36	C	43.09	38	690	1875	8.8	132	368	9	7720kb
R/W$_{K\geq 1}$	24	22	ME	9m31s	118	16402	3842	1	29119	1936	0.3	34008Kb

Fig. 31. Experiments using the symbolic algorithm keeping the CSTs in normal form and enriched with the structural heuristics.

Case-study	P	T	VP	EX(sec)	NI	Mem
Mesh2x2$_{K\geq 1}$	32	32	ME	144.65	15	948kb
PNCSA	31	36	SL	>319	11*	>500MB
PNCSA	31	36	C	>325	8*	>500MB
R/W$_{K\geq 1}$	24	22	ME	> 1058.04	15*	>500MB

Fig. 32. Experiments using HyTech. For the Mesh3x2$_{K\geq 1}$, we could not run HyTech because this example has too many variables.

11.5 Comparison with finite-state verification.

Several of the parameterized systems that we have verified are finite-state systems when the initial markings is fixed. Those instantiated systems can them be verified using the forward reachability algorithm of finite-state model checkers. We have run several examples with GreatSPN [10] (a state of the art model checker for finite-state Petri nets). In all the cases, even for small values of the parameters (small numbers of processes), the CST verification of the parameterized problem is much more efficient. For example, when run on the Mesh example with $K = 2$ (only two processes), GreatSPN takes $46.15s$ to compute the $200,544$ reachable states and is not able to compute the reachability set for $K = 3$ (three processes). In comparison, we were able to verify the Mesh for any number of processes (K left as a parameter) in $0.8s$. As foreseen by Bultan in [8], in most of the cases proving a parameterized property turns out to be more efficient than proving its finite-state instances.

12 Conclusions and Related Works

Parameterized verification has recently attracted a lot of attention in the computer aided verification community. When a counting abstraction is applicable to a parameterized verification problem, (infinite-state) Petri nets are a very useful abstract model. Therefore the need for an efficient analysis of infinite-state Petri nets is an important issue for proposing practically applicable verification methods for parameterized systems. In this paper, we have proposed an new efficient data structure called CSTs tailored to the representation of infinite upward-closed set of integer vectors and thus of upward-closed set of Petri net markings.

We have studied in details the theoretical complexities of the problems related to the symbolic manipulation of upward-closed sets of integer vectors represented as CSTs.We have shown that the ability of CSTs to represent compactly complex upward-closed sets of integer vectors, i.e. with a large numbers of minimal elements, has a theoretical cost: several useful operations have no polynomial time algorithms unless P= NP.

We have shown that CSTs can be used to efficiently solve coverability problems of practically interesting infinite-state Petri nets with a parametric initial marking. The algorithm compute symbolically the backward reachable states of the Petri net from an upward-closed set of markings. We have also established that structural invariants can be very effective for cutting the backward search space and allowed us to verify all the examples that we have considered with dramatically better execution time and more importantly to solve problems that are not in the reach of the procedure without that heuristic.

This paper includes and extends with proofs and new experiments the preliminary work presented in [14,15].

12.1 Other data structures for parameterized verification.

Other efficient data structures have been proposed for the symbolic representation of infinite sets of integer values.

Numerical Decision Diagrams (NDD) [4] can be used to represent any set of vectors of integers (as well as real) numbers definable in the arithmetic of Presburger. NDDs are thus fairly general and can be used to define any upward-closed set. They are thus a possible candidate as a data structure for the backward analysis of infinite-state Petri nets. As the NDD is a very expressive data structure, it remains to be tested if it will be usable to encode very complex upward-closed sets. We have shown in the previous subsection that a tool like HyTech that uses polyhedra, another fairly expressive data-structure, encounter rapidly the symbolic state explosion problem when apply to medium size infinite-state Petri nets.

Difference Decision Diagrams (DDDs) [32] has been introduced as an efficient data structure to represent large disjunction of a special class of polyhedra encountered during the verification of timed automata. Again this structure can be used to define infinite set of vectors of integer (as well as real) numbers. The DDDs are able to represent all sets definable by boolean combination of formula of the form : $x - y \sim c$ and $x \sim c$ where x, y are variable that ranges over the integer numbers, c is an integer constant, and $\sim \in \{<, \leq, =, \geq, >\}$.

Differently from NDDs and DDDs, the structure of CST provide for several optimizations (e.g. working at the level of single layers, or using simulation-based reductions) that seem to perform well on practical examples. In order to have a significant comparison of the three data structures, we are currently working on a common software platform in which to run the same set of examples by taking the same set of optimizations and assumptions. Note, in fact, that in our experiments we exploited several domain specific optimizations (e.g. structural heuristics). We are currently investigating the applicability of similar optimizations to the other data structures. The results of this comparison will be made available in a separate work.

References

1. P. A. Abdulla, K. Čerāns, B. Jonsson, and Y.-K. Tsay. General Decidability Theorems for Infinite-state Systems. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS'96)*, pages 313–321. IEEE Computer Society Press, 1996.
2. M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. Series in Parallel Computing. John Wiley & Sons, 1995.
3. B. Bérard, and L. Fribourg. Reachability Analysis of (Timed) Petri Nets Using Real Arithmetic. In *Proceedings of the 10th International Conference on Concurrency Theory (CONCUR'99)*, volume 1664 of *LNCS*, pages 178–193. Springer, 1999.
4. B. Boigelot, P. Wolper. Verifying systems with infinite but regular state space. In *Proceedings of the 10th International Conference on Computer Aided Verification (CAV'98)*, volume 1427 of LNCS, pages 88–97. Springer, 1998.
5. A. Bouajjani and R. Mayr. Model Checking Lossy Vector Addition Systems. In *Proceedings of the 16th Annual Symposium on Theoretical Aspects of Computer Science (STACS'99)*, volume 1563 of LNCS, pages 323–333. Springer, 1999.
6. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the 5th IEEE Symposium on Logic in Computer Science (LICS'90)*, pages 428–439. IEEE Computer Society Press, 1990.
7. T. Bultan, R. Gerber, and W. Pugh. Symbolic Model Checking of Infinite-state Systems using Presburger Arithmetics. In *Prococeedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of LNCS, pages 400–411. Springer 1997.
8. T. Bultan. BDD vs. Constraint-Based Model Checking: An Experimental Evaluation for Asynchronous Concurrent Systems. In *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, volume 1785 of LNCS, pages 441–455. Springer, 2000.
9. K. Čerāns. Deciding Properties of Integral Relational Automata. In *Proceedings of the 21st International Colloquium on Automata and Languages for Programming (ICALP 94)*, volume 820 of LNCS, pages 35–46. Springer, 1994.
10. G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo. GreatSPN 1.7: Graphical Editor and Analyzer for Timed and Stochastic Petri Nets. *Performance Evaluation*, 24(1-2), pages 47–68, 1995.
11. G. Ciardo. Petri nets with marking-dependent arc multiplicity: properties and analysis. In *Proceeding of the 15th International Conference on Applications and Theory of Petri Nets (ICATPN 94)*, volume 815 of LNCS, pages 179–198. Springer, 1994.
12. G. Delzanno, J. Esparza, and A. Podelski. Constraint-based Analysis of Broadcast Protocols. In *Proceedings of the 13th Annual Conference of the European Association for Computer Science Logic (CSL'99)*, volume 1683 of LNCS, pages 50–66. Springer, 1999.
13. L.E. Dickson. Finiteness of the odd perfect and primitive abundant numbers with r distinct prime factors. In *American Journal of Mathematics*, 35: 413–422. Johns Hopkins University Press, 1913.
14. G. Delzanno, and J.-F. Raskin. Symbolic Representation of Upward Closed Sets. In *Proceeding of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, volume 1785 of LNCS, pages 426–440, 2000. Also appeared as Technical Report MPI-I-1999-2-007, Max-Planck-Institut für Informatik, Saarbrücken,November 1999.

15. G. Delzanno, J.-F. Raskin, and L. Van Begin. Attacking Symbolic State Explosion. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV 2001)*, volume 2102 of LNCS, pages 298–310. Springer, 2001. Also appeared as Technical Report 441, Universitè Libre de Bruxelles, January 2001.
16. E. A. Emerson and K. S. Namjoshi. On Model Checking for Non-deterministic Infinite-state Systems. In *Proceedings of the 13th Annual Symposium on Logic in Computer Science (LICS '98)*, pages 70–80. IEEE Computer Society Press, 1998.
17. J. Esparza, A. Finkel, and R. Mayr. On the Verification of Broadcast Protocols. In *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 352–359. IEEE Computer Society Press, 1999.
18. J. Esparza and S. Melzer. Verification of safety properties using integer programming: Beyond the state equation. *Formal Methods in System Design*, 16:159–189. Kluwer Academic Publishers, 2000.
19. A. Finkel. Reduction and Covering of Infinite Reachability Trees. *Information and Computation* 89(2), pages 144–179. Academic Press, 1990.
20. A. Finkel. The minimal coverability graph for Petri nets. In *Proceedinfs of Advances in Petri Nets '93*, volume 674 of LNCS, pages 210–243. Springer, 1993.
21. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science* 256 (1-2):63–92. Elsevier, 2001.
22. R. Fraïssé Theory of relations / Roland Fraïssé; with an appendix by Norbert Sauer. Studies in logic and the foundations of mathematics, Elsevier, 2000.
23. S. M. German, A. P. Sistla. Reasoning about Systems with Many Processes. *Journal of ACM* 39(3): 675–735, 1992.
24. B. Grahlmann. The PEP Tool. In *Proceedings of the 9th Conference on Computer Aided Verification (CAV'97)*, volume 1254 of LNCS, pages 440–443. Springer, 1997.
25. R. M. Karp and R. E. Miller. Parallel Program Schemata. *Journal of Computer and System Sciences*, 3, pages 147-195. Academic Press, 1969.
26. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, E. Shahar. Symbolic Model Checking with Rich Assertional Languages. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97)*, volume 1254 of LNCS, pages 424–435. Springer, 1997.
27. D. S. Johnson. A Catalog of Complexity Classes. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A, Algorithm and Complexity*, Elsevier, 1990.
28. M. R. Henzinger, T. A. Henzinger, P. K. Kopke. Computing Simulations on Finite and Infinite Graphs. In *Proceedings of the 36th Annual IEEE Symposium on Foundations of Computer Science (FOCS'95)*, pages 453–462. IEEE Society Press, 1995.
29. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: a Model Checker for Hybrid Systems. In *Proceedings of the 9th Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 460–463. Springer, 1997.
30. A. Miner, and G. Ciardo. Efficient Reachability Set Generation and Storage using Decision Diagrams. In *Proceedings of th 20th International Conference on Applications and Theory of Petri Nets (ICATPN '99)*, volume 1639 of LNCS, pages 6–25. Springer, 1999.
31. N. M. Minsky. Finite and Infinite Machines. Prentice Hall, Englewood Cliffs, N.Y., 1967.
32. J. Møller, J. Lichtenberg, H. R. Andersen, H. Hulgaard. Difference Decision Diagrams. In *Proceeding of the 13th Annual Conference of the European Association for Computer Science Logic (CSL'99)*, volume 1683 of LNCS, pages 111–125. Springer, 1999.
33. W. Reisig. Petri Nets. An introduction. *EATCS Monographs on Theoretical Computer Science*, Springer, 1986.
34. M. Silva, E. Teruel, and J. M. Colom. Linear Algebraic and Linear Programming Techniques for Analysis of Place/Transition Net Systems. In *Proceedings of Advances in Petri Nets*, volume 1491 of LNCS, pages 308–309. Springer, 1998.
35. E. Teruel. *Structure Theory of Weighted Place/Transition Net Systems: The Equal Conflict Hiatus.* Ph.D. Thesis, University of Zaragoza, Zaragoza, Spain, 1994.
36. D. Zampuniéris. The Sharing Tree Data Structure: Theory and Applications in Formal Verification. Ph.D. Thesis. Facultés Universitaires Notre-Dame de la Paix, Namur, Belgium, 1997.
37. D. Zampuniéris, and B. Le Charlier. Efficient Handling of Large Sets of Tuples with Sharing Trees. In *Proceedings of the Data Compressions Conference (DCC'95)*, page 428, 1995.

A Algorithms for adjusting conditions 5 and 6

```
procedure Adjust_Condition5(S : pre−CST)
    for each i : 0 to k (top to bottom) do
        for each node n of N_i^S do
            newsucc ← ∅;
            for each value c such that ∃n' ∈ succ^S(n) : val^S(n') = c do
                Construct a new node n'' with val^S(n'') = c;
                for each n' ∈ succ^S(n) such that val^S(n') = c do
                    succ^S(n'') ← succ^S(n'') ∪ succ^S(n');
                endfor
                if ∃n' ∈ N_{i+1}^S : val^S(n') = c ∧ SameSons(n', n'') then
                    let m be such a node and newsucc ← newsucc ∪ {m};
                else
                    newsucc ← newsucc ∪ {n''};
                    N_{i+1}^S ← N_{i+1}^S ∪ {n''};
                endif
            endfor
            succ^S(n) ← newsucc;
        endfor
        for each node n ∈ N_{i+1}^S do
            if ∄n' ∈ N_i^S : n ∈ succ^S(n') then
                remove n from N_{i+1}^S;
            endif
        endfor
    endfor
endprocedure
```

Fig. 33. Algorithm that ensures condition 5

```
procedure Adjust_Condition6(S : pre−CST)
    for each n ∈ N^S do
        remove(n) ← false;
    endfor
    for each i : k to 1 (bottom − up) do
        for each node n ∈ N_i^S with remove(n) = false do
            for each node n' ∈ N_i^S (n' ≠ n) such that val^S(n) = val^S(n')
                                                        ∧SameSons(n, n') do
                for each node n'' such that n' ∈ succ^S(n'') do
                    succ^S(n'') \ {n'};
                    succ^S(n'') ∪ {n};
                endfor
                remove(n') ← true;
            endfor
        endfor
    endfor
    for each n ∈ N_i^S such that remove(n) = true do
        N_i^S \ {n};
    endfor
endprocedure
```

Fig. 34. Algorithm that ensures condition 6