Symbolic Data Structure for sets of *k*-uples of integers

Pierre Ganty[1], Cédric Meuter[1], Laurent Van Begin[1], Gabriel Kalyon[1], Jean-François Raskin[1], and Giorgio Delzanno[2]

[1] Département d'Informatique,
Université Libre de Bruxelles
[2] Dipartimento di Informatica e Scienze dell'Informazione,
Università degli Studi di Genova

Abstract. In this document we present a new symbolic data structure dedicated to the manipulation of (possibly infinite) sets of k-uples over integers, initially introduced in [Gan02]. This new data structure called *Interval Sharing Tree* (IST), is based on sharing trees [ZL95] where each node is labelled with an interval of integers. We present symbolic algorithm on IST for standard set operations and also introduce some specific operation that can be useful in the context of model-checking.

1 Introduction

Many systems like multi-threaded programs and communication protocols are very complex to devise, hence error prone. For almost three decades a lot of research has been done in order to formalize and prove or refute properties on such systems. Given a formal description $\mathcal{M}$ of the system, we apply *model checking* methods [CGP99] to verify automatically that some properties hold on $\mathcal{M}$. Those automatic methods manipulate (possibly) infinite sets of configurations of $\mathcal{M}$. In model checking one problem we face off is the *(symbolic) state explosion problem*. This problem corresponds to a blow up in the size of the sets of (symbolic) configurations being manipulated. One way to overcome that problem is to use a compact symbolic representation of those sets.

In our context, configurations are tuples of integers. (infinite) sets of configurations are represented by tuples of intervals that correspond to symbolic configurations. In this paper, we define *Interval Sharing Tree* (IST) [Gan02] that allows to efficiently represent a finite set of tuples of intervals, hence a (infinite) set of tuples of integers.

ISTs are directed acyclic graph-based data structures, where nodes are labelled with intervals of integers. This symbolic data structure benefits from several advantages. First, algorithms can be defined to manipulate ISTs symbolically (i.e. without exploring all path of the graph). Second, the size of an IST $\mathcal{I}$ may be exponentially smaller than the size of the set of k-uples it represent. Finally, an IST $\mathcal{I}$ can be (semantically) non-deterministic whereas other approaches facing to the state explosion problem traditionally use deterministic data structures (e.g. sharing trees [ZL95], BDD [CGP99], ...), thus allowing $\mathcal{I}$ to be more compact.

The remainder of this document is structured as follows. First, in sec. 2, we introduce interval sharing trees and their semantics. We follow up, in sec. 3, by presenting symbolic algorithms for manipulating this data structure. Finally, in sec. 4, we conclude by presenting a summary of all operations and their complexity.

2 Interval Sharing Tree

In the following, $\mathbb{Z}$ denotes the set of integers and $\mathbb{Z}^\infty = \mathbb{Z} \cup \{-\infty, +\infty\}$. The classical order over integer is extended to $\mathbb{Z}^\infty$ in the usual way: $\forall c \in \mathbb{Z} : -\infty < c < +\infty$. An interval is a pair $\langle a, b\rangle \in \mathbb{Z}^\infty \times \mathbb{Z}^\infty$ such that $a \leq b$, noted $[a, b]$. As usual, an interval $\langle a, b\rangle$ denotes the set $\{c \in \mathbb{Z} \mid a \leq c \leq b\}$. Hence, $c \in [a, b]$ means that c is in the set corresponding to $[a, b]$. We note $\mathbb{I}$ the set of all intervals. Moreover, we note $(a, b] = [a, b] \setminus \{a\}$ and $[a, b) = [a, b] \setminus \{b\}$.

Definition 1 (Interval Sharing Tree). *An interval sharing tree (IST in short) $\mathcal{I}$, is a labelled directed rooted acyclic graph $\langle N, \iota, \mathsf{succ}\rangle$ where:*

- $N = N^0 \cup N^1 \cup N^2 \cup ... \cup N^k \cup N^{k+1}$ *is the finite set of nodes, partitioned into $k+2$ disjoint subsets N^i called layers with $N^0 = \{root\}$ and $N^{k+1} = \{end\}$*
- $\iota : N \mapsto \mathbb{I} \cup \{\top, \bot\}$ *is the labelling function such that $\iota(n) = \top$ (resp. $\bot$) if and only if $n = root$ (resp. end).*
- $\mathsf{succ} : N \mapsto 2^N$ *is the successor function such that:*
  (i) $\mathsf{succ}(end) = \emptyset$
  (ii) $\forall i \in [0, k], \forall n \in N^i : \mathsf{succ}(n) \subseteq N^{i+1} \wedge \mathsf{succ}(n) \neq \emptyset$
  (iii) $\forall n \in N, \forall n_1, n_2 \in \mathsf{succ}(n) : (n_1 \neq n_2) \Rightarrow (\iota(n_1) \neq \iota(n_2))$
  (iv) $\forall i \in [0, k], \forall n_1 \neq n_2 \in N^i : (\iota(n_1) = \iota(n_2)) \Rightarrow (\mathsf{succ}(n_1) \neq \mathsf{succ}(n_2))$

In other words, an IST is a directed acyclic graph where the nodes are labelled with intervals except for two special nodes (*root* and *end*), such that (i) the *end* node has no successors, (ii) all nodes from layer i have their successors in layer $i+1$, (iii) a node cannot have two successors with the same label, (iv) two nodes with the same label in the same layer do not have the same successors. Condition (iii) ensures maximal prefix sharing, i.e. the IST is syntactically deterministic. It does not mean that an IST is *semantically deterministic*, i.e. a node may have two distinct successors labeled with intervals that are not disjoint. Condition (iv) ensures maximal suffix sharing. If a graph satisfies all condition of definition 1 by those two condition, it is called a pre-IST.

A path of an IST is a k-uple $\langle n_1, n_2, ..., n_k\rangle$ such that $n_1 \in \mathsf{succ}(root), end \in \mathsf{succ}(n_k)$ and $\forall i \in [1, k) : n_{i+1} \in \mathsf{succ}(n_i)$. Given an IST $\mathcal{I}$, we note $\mathsf{path}(\mathcal{I})$, the set of path of $\mathcal{I}$. The semantics of an IST $\mathcal{I}$, noted $[\![\mathcal{I}]\!]$ is the set $\{\overline{x} \in \mathbb{Z}^k \mid \exists\langle n_1, ..., n_k\rangle \in \mathsf{path}(\mathcal{I}), \forall i \in [1, k] : x_i \in \iota(n_i)\}$.

3 Operations on IST

In this section we present the operations on IST. Those operation are symbolic, i.e. they work on the tree instead of the set of k-uples that are represented. We start in sec. 3.1 by introducing some preliminary operations that will later on. Then, in sec. 3.2, we introduce basic set operations, i.e. emptiness test, membership test, union, intersection and complement. Finally, in sec. 3.3, we present some operations that are useful in the context of model-checking, namely substitution and downward closure.

3.1 Preliminary operations

Prefix sharing Given a pre-IST $\mathcal{I}$ with k layers, we want to modify $\mathcal{I}$ so that it satisfies condition (iii) of definition 1. The idea is to traverse the sharing tree *layer-by-layer* and to determinize syntactically the nodes of each layer. This algorithm has a wost-case time complexity of $O(2^{|\mathcal{I}|})$. Note however, that this algorithm returns a pre-IST satisfying condition iii, but not necessarily condition (iv).

Algorithm 1: $\mathsf{enforcePrefixSharing}(\mathcal{I}, k)$

> **data**: $\mathcal{I} = \langle N, \iota, \mathsf{succ}\rangle$, $k \in \mathbb{N}$
> **post**: $\forall n \in N, \forall n_1, n_2 \in \mathsf{succ}(n) : (n_1 \neq n_2) \Rightarrow (\iota(n_1) \neq \iota(n_2))$
> **begin**
>   **for** $i \leftarrow 0$ **to** k **do**
>   **forall** $n \in N^i$ **do**
> $S \leftarrow \emptyset$
>   **forall** $I \in \mathbb{I}$ **s.t.** $\exists n' \in \mathsf{succ}(n) : \iota(n') = I$ **do**
>   let n'' be a new node
> $\mathsf{succ}(n'') \leftarrow \emptyset$, $\iota(n'') \leftarrow I$
>   **forall** $n' \in \mathsf{succ}(n)$ **s.t.** $\iota(n') = I$ **do**
> $\mathsf{succ}(n'') \leftarrow \mathsf{succ}(n'') \cup \mathsf{succ}(n')$
> $N^{i+1} \leftarrow N^{i+1} \cup \{n''\}$
> $S \leftarrow S \cup \{n''\}$
> $\mathsf{succ}(n) \leftarrow S$
> **end**

Suffix sharing Given a pre-IST $\mathcal{I}$ with k layers, we want to transom $\mathcal{I}$ so that it satisfies condition (iv) of definition 1. The idea is to traverse the sharing tree *layer-by-layer* (bottom-up). When we encounter two nodes in same layer with same label and with the same set of sons, we delete one of them (the other one takes its place). Note that this operation preserves condition iii. Therefore, if one wants to enforce both conditions, one must start with $\mathsf{enforcePrefixSharing}()$.

Algorithm 2: enforceSuffixSharing($\mathcal{I}, k$)

data: $\mathcal{I} = \langle N, \iota, \mathsf{succ} \rangle$, $k \in \mathbb{N}$
post: $\forall i \in [0, k], \forall n_1 \neq n_2 \in N^i : (\iota(n_1) = \iota(n_2)) \Rightarrow (\mathsf{succ}(n_1) \neq \mathsf{succ}(n_2))$

```
begin
    for i ← k downto 1 do
        forall n ∈ N^i do
            forall n′ ∈ N^i \ {n} s.t. ι(n) = ι(n′) ∧ succ(n) = succ(n′) do
                forall n″ ∈ N^{i−1} s.t. n′ ∈ succ(n″) do
                    succ(n″) ← (succ(n″) \ {n′}) ∪ {n}
                N^i ← N^i \ {n′}
end
```

Algorithm 3: determinize($\mathcal{I}, k$)

data: $\mathcal{I} = \langle N, \iota, \mathsf{succ} \rangle$, $k \in \mathbb{N}$
post: $\forall n \in N, \forall n_1 \neq n_2 \in \mathsf{succ}(n) : \iota(n_1) \cap \iota(n_2) = \emptyset$

```
begin
    for i ← 0 to k do
        forall n ∈ N^i s.t. ∃n_1 ≠ n_2 ∈ succ(n) : ι(n_1) ∩ ι(n_2) ≠ ∅ do
            let n′ be a new node
            ι(n′) ← ι(n)
            forall n″ ∈ N s.t. n ∈ succ(n″) do
                succ(n″) ← succ(n″) ∪ {n′} \ {n}
            L ← {ι(n″) | n″ ∈ succ(n)}
            let L′ ∈ 2^𝕀 be the smallest partition of ∪_{I∈L} I
            forall I′ ∈ L′ do
                let s be a new node
                forall t ∈ succ(n) s.t. I′ ⊆ ι(t) do
                    succ(s) ← succ(s) ∪ succ(t)
                ι(s) ← I′
                succ(n′) ← succ(n′) ∪ {s}
                N^{i+1} ← N^{i+1} ∪ {s}
            N^i ← N^i ∪ {n′} \ {n}
    enforceSuffixSharing(I, k)
end
```

Determinization Given an IST $\mathcal{I}$ with k layers, we want to eliminate from $\mathcal{I}$ the semantical non-determinism that can arise in the successors of a given node while preserving $[\![\mathcal{I}]\!]$. After the determinization of a sharing tree, the intevals defined by the successors of any node must be disjoint. The idea of the algorithm is to traverses the sharing tree *layer-by-layer* and for each node n, compute a new set of disjoint intervals. Then, we create a node s for each such interval and we compute the successors of s, i.e. the successors of nodes that belongs to $\mathsf{succ}(n)$

and that have a label containing the label of s. This algorithm has a worst-case complexity of $O(2^{|\mathcal{I}|})$.

3.2 Set operations

Emptiness Given an IST $\mathcal{I}$ with k layers, we want to decide if $[\![\mathcal{I}]\!] = \emptyset$ holds. The algorithm for that is very simple. Indeed, $[\![\mathcal{I}]\!] = \emptyset$ if and only if $\mathsf{succ}(root) = \emptyset$. This algorithm has a worst-case time complexity of $O(1)$.

Algorithm 4: $\mathsf{empty}(\mathcal{I}, k)$

```
data    : I = ⟨N, ι, succ⟩, k ∈ ℕ
returns: true if and only if [[I]] = ∅
begin
    return succ(root) = ∅
end
```

Membership Given an IST $\mathcal{I}$ with k layers and a k-uple $\overline{x} = \langle x_1, \ldots, x_k \rangle \in \mathbb{Z}^k$, we want to determine if $\overline{x} \in [\![\mathcal{I}]\!]$ holds. For that, we present an algorithm for that problem adapted from *Covering Sharing Trees* [Van03,DRV04] that has a worst-case time complexity $O(|\mathcal{I}|)$. The algorithm is based on a *layer-by-layer* traversal of the graph. We first mark the *root* node of $\mathcal{I}$. Then for each layer $i \in [1, k+1]$, we mark all nodes $n \in N_i$ such that $x_i \in \iota(n)$ and such that n has a marked predecessor. At the end of this procedure, $\overline{x} \in \mathcal{I}$ holds if and only if the *end* node has at least one marked predecessor.

Algorithm 5: $\mathsf{memberOf}(\mathcal{I}, k, \overline{x})$

```
data    : I = ⟨N, ι, succ⟩, k ∈ ℕ, x̄ ∈ ℤ^k
returns: true if and only if x̄ ∈ [[I]] holds
begin
    M ← {root}
    for i ← 1 to k do
        forall n ∈ N^i do
            if (x_i ∈ ι(n)) ∧ (∃n' ∈ M : n ∈ succ(n')) then
                M ← M ∪ {n}
    return (∃n ∈ M : end ∈ succ(n))
end
```

Intersection Given two IST $\mathcal{I}_1$, $\mathcal{I}_2$ with k layers, we want to compute an IST $\mathcal{I}$ such that $[\![\mathcal{I}]\!] = [\![\mathcal{I}_1]\!] \cap [\![\mathcal{I}_2]\!]$. The algorithm is adapted from the the algorithm computing the synchronized product of finite automata. Each node in this product is a pair of node from both $\mathcal{I}_1$ and $\mathcal{I}_2$. In this particular case, the interval of a node $\langle n_1, n_2\rangle$ in the product is the intersection of the interval of n_1 and n_2. Unfortunately, this can introduce some prefix and/or suffix redundancy, thus violating conditions (*iii*) and/or (*iv*) of definition 1. Therefore, we must enforce those two conditions in the algorithm to turn the resulting pre-IST into an IST. For this reason, this algorithm has a worst-case time complexity of $O(2^{|\mathcal{I}_1|\times|\mathcal{I}_2|})$. It was proven in [Van03, thm. 20, 21] that for *Covering Sharing Trees*, which is a subclass of IST, no polynomial algorithm exists for that problem.

Algorithm 6: $\mathsf{intersection}(\mathcal{I}_1, \mathcal{I}_2, k)$

```
data    : I1 = ⟨N1, ι1, succ1⟩, I2 = ⟨N2, ι2, succ2⟩, k ∈ ℕ
returns: I s.t. ⟦I⟧ = ⟦I1⟧ ∩ ⟦I2⟧
begin
    I ← ⟨N^0 ∪ ... ∪ N^{k+1}, ι, succ⟩
    for i ← 0 to k do
        N^i ← ∅
    N^{k+1} ← {⟨end1, end2⟩}
    ι(⟨end1, end2⟩) ← ⊥
    for i ← k downto 1 do
        forall ⟨n1, n2⟩ ∈ N1^k × N2^k s.t. ι1(n1) ∩ ι2(n2) ≠ ∅ do
            ι(⟨n1, n2⟩) ← ι1(n1) ∩ ι2(n2)
            succ(⟨n1, n2⟩) ← N^{k+1} ∩ (succ1(n1) × succ2(n2))
            if succ(⟨n1, n2⟩) ≠ ∅ then
                N^k ← N^k ∪ {⟨n1, n2⟩}
    N^0 ← {⟨root1, root2⟩}
    ι(⟨root1, root2⟩) ← ⊤
    succ(⟨root1, root2⟩) ← N^1
    enforcePrefixSharing(I, k)
    enforceSuffixSharing(I, k)
    return I
end
```

Union Given two IST $\mathcal{I}_1$, $\mathcal{I}_2$ with k layers, we want to compute an IST $\mathcal{I}$ such that $[\![\mathcal{I}]\!] = [\![\mathcal{I}_1]\!] \cup [\![\mathcal{I}_2]\!]$. This can be done using the algorithm to compute the union of two sharing trees defined in [ZL95]. This algorithm uses two recursive functions $\mathsf{recUnion}()$ and $\mathsf{recCopy}()$. The function $\mathsf{recUnion}()$ computes the union of a sub-tree of $\mathcal{I}_1$ rooted in n_1 and a sub-tree of $\mathcal{I}_2$ rooted in n_2. The roots of these sub-trees must have the same labels. The function $\mathsf{recCopy}()$ copies a sub-tree of $\mathcal{I}_1$ or $\mathcal{I}_2$ in $\mathcal{I}$. Those two function make use of a hashing table T to remember previous computations. When the union of two sub-trees rooted respectively in n_1 and n_2 is computed and added in $\mathcal{I}$, the resulting node is stored in T, indexed by the pair $\langle n_1, n_2\rangle$. Similarly, when a sub-tree rooted in n is copied in $\mathcal{I}$, the copied node is stored in T, indexed by the pair $\langle n, n\rangle$. This is essential in order treat every pair of nodes only once, thus allowing a polynomial complexity. Note that condition iv does not necessarily holds on the result and must therefore be enforced. This algorithm has a worst-case time complexity of $O((|\mathcal{I}_1| + |\mathcal{I}_2|)^3)$.

Algorithm 7: $\mathsf{recUnion}(n_1, n_2, i)$

```
begin
    if T(n1, n2) is defined then
        n ← T(n1, n2)
    else
        let n be a new node
        ι(n) ← ι1(n1)
        N^i ← N^i ∪ {n}
        forall n'1 ∈ succ1(n1) do
            if ∃n'2 ∈ succ2(n2) s.t. ι1(n'1) = ι2(n'2) then
                succ(n) ← succ(n) ∪ {recUnion(n'1, n'2, i + 1)}
            else
                succ(n) ← succ(n) ∪ {recCopy(n'1, 1, i + 1)}
        forall n'2 ∈ succ2(n2) do
            if ∀n'1 ∈ succ1(n1) : ι1(n'1) ≠ ι2(n'2) then
                succ(n) ← succ(n) ∪ {recCopy(n'2, 2, i + 1)}
        T(n1, n2) ← n
    return n
end
```

Algorithm 8: $\mathsf{recCopy}(n, x, i)$

```
begin
    if T(n, n) is defined then
        n' ← T(n, n)
    else
        let n' be a new node
        ι(n') ← ιx(n)
        Ni ← Ni ∪ {n'}
        forall n'' ∈ succx(n) do
            succ(n') ← succ(n') ∪ {recCopy(n'', x, i + 1)}
        T(n, n) ← n'
    return n'
end
```

Algorithm 9: $\mathsf{union}(\mathcal{I}_1, \mathcal{I}_2, k)$

data: $\mathcal{I}_1 = \langle N_1, \iota_1, \mathsf{succ}_1 \rangle$, $\mathcal{I}_2 = \langle N_2, \iota_2, \mathsf{succ}_2 \rangle$, $k \in \mathbb{N}$
returns: $\mathcal{I}$ **s.t.** $[\![\mathcal{I}]\!] = [\![\mathcal{I}_1]\!] \cup [\![\mathcal{I}_2]\!]$
begin
- $\mathcal{I} \leftarrow \langle N^0 \cup ... \cup N^{k+1}, \iota, \mathsf{succ} \rangle$
- **forall** $i \leftarrow 0$ **to** k **do**
  - $N^i \leftarrow \emptyset$
- $\mathsf{recUnion}(root_1, root_2, 0)$
- $\mathsf{enforceSuffixSharing}(\mathcal{I}, k)$
- **return** $\mathcal{I}$

end

Algorithm 10: $\mathsf{complement}(\mathcal{I}, k)$

data : $\mathcal{I} = \langle N, \iota, \mathsf{succ} \rangle$, $k \in \mathbb{N}$
returns: $\mathcal{I}_1$ such that $[\![\mathcal{I}_1]\!] = \mathbb{Z}^k \setminus [\![\mathcal{I}]\!]$
begin
- $\mathcal{I}_1 \leftarrow \mathcal{I}$
- $\mathsf{determinize}(\mathcal{I}_1)$
- $M \leftarrow N_1^{k-1}$
- **for** $i \leftarrow 0$ **to** k **do**
  - **forall** $n \in N_1^i$ **do**
    - $L \leftarrow \{\iota_1(n') \mid n' \in \mathsf{succ}_1(n)\}$
    - let $L' \in 2^{\mathbb{I}}$ be the smallest partition of $\mathbb{Z} \setminus (\cup_{I \in L} I)$
    - **forall** $I' \in L'$ **do**
      - let n' be a new node
      - $\iota_1(n') \leftarrow I'$, $\mathsf{succ}_1(n) \leftarrow \mathsf{succ}_1(n) \cup \{n'\}$
      - $N_1^{i+1} \leftarrow N_1^{i+1} \cup \{n'\}$
- **forall** $n \in M$ **do**
  - $\mathsf{succ}_1(n) \leftarrow \emptyset$;
- **forall** $n \in N_1^{k-1} \setminus M$ **do**
  - $\mathsf{succ}_1(n) \leftarrow end_1$
- **for** $i \leftarrow k$ **downto** 1 **do**
  - **forall** $n \in N^i$ **do**
    - **if** $\mathsf{succ}_1(n) = \emptyset$ **then**
      - $N_1^i \leftarrow N_1^i \setminus \{n\}$
      - **forall** $n' \in N_1^i$ **s.t.** $n \in \mathsf{succ}_1(n')$ **do**
        - $\mathsf{succ}_1(n') \leftarrow \mathsf{succ}_1(n') \setminus \{n\}$
- $\mathsf{enforceSuffixSharing}(\mathcal{I}_1, k)$
- **return** $\mathcal{I}_1$

end

Complement Given an IST $\mathcal{I}$ with k layers, we want to compute an IST $\mathcal{I}_1$ such that $[\![\mathcal{I}_1]\!] = \mathbb{Z}^k \setminus [\![\mathcal{I}]\!]$. First, we take a copy $\mathcal{I}_1$ of $\mathcal{I}$, and determinize it. Then, for each node n of $\mathcal{I}_1$, we complement the set of intervals defined by the successors of n to obtain a partition of $\mathbb{Z}$. We create a new node for each of these new intervals and add these new nodes to $\mathsf{succ}(n)$. In fact, these new nodes allow to consider the k-uples that do not belong to $[\![\mathcal{I}]\!]$. To delete the k-uples that belong to $[\![\mathcal{I}]\!]$, we delete the edges between the nodes of the layer N_1^k that were copied from $\mathcal{I}_1$ and the node end_1. Finally, we delete from the IST all path not ending in end_1.

3.3 Dedicated operations

We present in this section operations on IST useful in the framework of Model checking. Those operations naturally appear when designing verification algorihms for Petri nets and their monotonic extensions [DRV04,FRSV02].

Substitution Given an IST $\mathcal{I}$ with k layers, we want to compute an IST $\mathcal{I}_1$ where a certain variable x_i is replaced by $x_i + d$ with $d \in \mathbb{Z}$. Formally, we want that $[\![\mathcal{I}_1]\!] = [\![\mathcal{I}]\!]_{[x_i+d/x_i]} = \{\langle x_1, ..., x_i + d, ..., x_k\rangle \mid \overline{x} \in \mathcal{I}\}$. This operation can be done very simply, by shifting all interval of layer i. Note that this transformation does not violate condition (iv) of (iii).

Algorithm 11: $\mathsf{substitution}(\mathcal{I}, k, i, d)$

data : $\mathcal{I} = \langle N, \iota, \mathsf{succ}\rangle$, $k \in \mathbb{N}$, $i \in [1, k]$, $d \in \mathbb{Z}$
returns: $\mathcal{I}_1$ **s.t.** $[\![\mathcal{I}_1]\!] = [\![\mathcal{I}]\!]_{[x_i+d/x_i]}$
begin
- $\mathcal{I}_1 \leftarrow \langle N, \iota_1, \mathsf{succ}\rangle$
- **forall** $n \in N \setminus N^i$ **do**
  - $\iota_1(n) \leftarrow \iota(n)$
- **forall** $n \in N^i$ **do**
  - $\langle l, u\rangle \leftarrow \iota(n)$
  - $\iota_1(n) \leftarrow \langle l + d, u + d\rangle$
- **return** $\mathcal{I}_1$

end

Downward Closure Given a subset $S \subseteq \mathbb{Z}^k$, the downward closure of S, noted $\downarrow S$ is the set with k-uple smaller than a k-uple of S. Formally, $\downarrow S = \{\overline{x} \in \mathbb{Z}^k \mid \exists \overline{x}' \in S\ \forall i \in [1, k] : x_i \leq x'_i\}$. Hence, given an IST $\mathcal{I}$ with k layers, we want to compute an IST $\mathcal{I}_1$ such that $[\![\mathcal{I}_1]\!] = \downarrow [\![\mathcal{I}]\!]$. The algorithm works as follows. The left bound of all nodes is set to $-\infty$. This transformation can introduce some prefix and/or suffix redundancy, thus violating conditions (iii) and/or (iv) of definition 1. Therefore, we must enforce those two conditions in the algorithm. The algorithm has a worst-time complexity $O(2^{|\mathcal{I}|})$.

Operation	Effect	Complexity
$\mathsf{enforcePrefixSharing}(\mathcal{I}, k)$	enforce condition (iii) on $\mathcal{I}$	$O(2^{\vert\mathcal{I}\vert})$
$\mathsf{enforceSuffixSharing}(\mathcal{I}, k)$	enforce condition (iv) on $\mathcal{I}$	$O((\vert\mathcal{I}\vert)^2)$
$\mathsf{determinize}(\mathcal{I}, k)$	semantically determinize $\mathcal{I}$	$O(2^{\vert\mathcal{I}\vert})$
$\mathsf{empty}(\mathcal{I}, k)$	test if $[\![\mathcal{I}]\!] = \emptyset$ holds	$O(1)$
$\mathsf{memberOf}(\mathcal{I}, k, \overline{x})$	test if $\overline{x} \in [\![\mathcal{I}]\!]$ holds	$O(\vert\mathcal{I}\vert)$
$\mathsf{union}(\mathcal{I}_1, \mathcal{I}_2, k)$	compute $\mathcal{I}$ s.t. $[\![\mathcal{I}]\!] = [\![\mathcal{I}_1]\!] \cup [\![\mathcal{I}_2]\!]$	$O((\vert\mathcal{I}_1\vert + \vert\mathcal{I}_2\vert)^3)$
$\mathsf{intersection}(\mathcal{I}_1, \mathcal{I}_2, k)$	compute $\mathcal{I}$ s.t. $[\![\mathcal{I}]\!] = [\![\mathcal{I}_1]\!] \cap [\![\mathcal{I}_2]\!]$	$O(2^{\vert\mathcal{I}_1\vert \times \vert\mathcal{I}_2\vert})$
$\mathsf{substitution}(\mathcal{I}, k, i, d)$	computes $\mathcal{I}_1$ s.t. $[\![\mathcal{I}_1]\!] = [\![\mathcal{I}]\!]_{[x_i+d/x_i]}$	$O(\vert\mathcal{I}\vert)$
$\mathsf{complement}(\mathcal{I}, k)$	compute $\mathcal{I}_1$ s.t. $[\![\mathcal{I}_1]\!] = \mathbb{Z}^k \setminus [\![\mathcal{I}]\!]$	$O(2^{\vert\mathcal{I}\vert})$
$\mathsf{downwardClosure}(\mathcal{I}, k)$	compute $\mathcal{I}_1$ s.t. $[\![\mathcal{I}_1]\!] = \downarrow[\![\mathcal{I}]\!]$	$O(2^{\vert\mathcal{I}\vert})$

Table 1. Operations on IST and their complexity

Algorithm 12: $\mathsf{downwardClosure}(\mathcal{I}, k)$

```
data    : I = ⟨N, ι, succ⟩, k ∈ ℕ
returns: I_1 s.t. [[I_1]] = ↓[[I]]
begin
    forall n ∈ N do
        ⟨l, u⟩ ← ι(n)
        ι_1(n) ← ⟨−∞, u⟩
    I_1 ← ⟨N, ι_1, succ⟩
    enforcePrefixSharing(I_1)
    enforceSuffixSharing(I_1)
    return I_1
end
```

4 Conclusion

In this paper, we defined a symbolic data structure to represent efficiently sets of k-uples of integers, called *Interval Sharing Trees*. We presented algorithms in order to manipulate this data structure symbolically. Those operations include traditional set operations as well as specific operation dedicated to *model-cheking*, and are summarized, along with complexity in tab. 1.

References

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[DRV04] Giorgio Delzanno, Jean-Francois Raskin, and Laurent Van Begin. Covering sharing trees: A compact data structure for parameterized verification. *Software Tools for Technology Transfer manuscript*, 2004.

[FRSV02] A. Finkel, J.-F. Raskin, M. Samuelides, and L. Van Begin. Monotonic Extensisions of Petri Nets : Forward and Backward Search Revisited. In *Proceedings of the 4th international workshop on verification of infinite-state systems (INFINITY 2002)*, volume 68 of *ENTCS*. Elsevier, 2002.

[Gan02] Pierre Ganty. Algorithmes et structures de données efficaces pour la manipulation de contraintes sur les intervalles. Master's thesis, Université Libre de Bruxelles, 2002.

[Van03] Laurent Van Begin. *Efficient Verification of Counting Abstractions for Parametric systems*. PhD thesis, Université Libre de Bruxelles, 2003.

[ZL95] Denis Zampunieris and Baudouin Le Charlier. Efficient handling of large sets of tuples with sharing trees. In *Proceedings of the 5th Data Compression Conference (DCC'95)*, page 428. IEEE Computer Society Press, 1995.