Sigma Decision Diagrams

BUCHS, Didier, HOSTETTLER, Steve Patrick

Abstract

Encoding and rewriting of large set of terms is very useful in a number of domains, such as model checking and theorem proving. The challenge of encoding and normalizing several billions of terms requires efficient ways of representing and manipulating them. Term Graph Rewriting is a well-known technique to share common sub-terms and thus to save both memory and processing time. However, this does not always fit well to the operational framework since it destroys the original structure and replaces it by a new one. This paper introduces a new kind of Decision Diagrams (DD), especially designed to handle set of terms in an efficient way. Based on the Set Decision Diagrams(SDD), an evolution of the well-known Binary Decision Diagrams(BDD), we propose the Sigma Decision Diagrams (ΣDD), a new approach to perform Term Rewriting on a set of terms in order to compute the image of that set efficiently.

Reference

Abstract

Encoding and rewriting of large set of terms is very useful in a number of domains, such as model checking and theorem proving. The challenge of encoding and normalizing several billions of terms requires efficient ways of representing and manipulating them. Term Graph Rewriting is a well-known technique to share common sub-terms and thus to save both memory and processing time. However, this does not always fit well to the operational framework since it destroys the original structure and replaces it by a new one. This paper introduces a new kind of Decision Diagrams (DD), especially designed to handle set of terms in an efficient way. Based on the Set Decision Diagrams (SDD), an evolution of the well-known Binary Decision Diagrams (BDD), we propose the Sigma Decision Diagrams ($\Sigma$DD), a new approach to perform Term Rewriting on a set of terms in order to compute the image of that set efficiently.

Keywords: Term Rewriting, Set Decision Diagrams, $\Sigma$ Decision Diagrams, Set of terms

1 Introduction

When performing model checking [1] using formalisms such as High Level Petri Nets [2,3] or the Chemical Abstract Machine [4] one has to manipulate large sets (usually billions) of terms expressing the state (or the set of states) of the system. This is a challenge both in terms of memory footprint and CPU consumption.

Term Graph Rewriting is a well-known approach to perform efficient rewriting. However, it alters the current term and thus the current state. This drawback becomes prohibitive when performing model checking since we need to keep all reachable states of the system. Therefore, we need another approach to rewrite sets of terms.

This paper proposes the Sigma Decision Diagrams ($\Sigma$DD), a new data structure that is an extension of the Set Decision Diagrams (SDD) [5] which are themselves an improvement of the Data Decision Diagrams (DDD) [6] and the well-known Binary Decision Diagrams (BDD) [7]. Roughly speaking, those structures handle sets of sequences of assignments in a symbolic way, by sharing common sub-graphs.

1 This project was partially funded by the COMEDIA project of the Hasler foundation, ManCom initiative project number 2107.

This paper is electronically published in Electronic Notes in Theoretical Computer Science URL: www.elsevier.nl/locate/entcs
Contrary to standard term graph rewriting, $\Sigma DD$ happens to preserve confluence w.r.t. term rewriting [8].

The SDD framework provides us a basis for performing Order-Sorted Term Rewriting [9,10] on sets of terms. It also enables us to share common sub-terms and rewriting steps reducing both the memory footprint and the processing time.

Ultimately the goal is to provide a complete framework based on $DDD$, $SDD$ and $\Sigma DD$, in order to perform model checking of High Level Petri Nets and particularly on Algebraic Petri Nets [2,3] and their extensions as described in [11].

The article is organized as follows: first we establish the prerequisites to encode a Term and the Rewriting Rules using the Decision Diagrams Framework formally. Then we define the $\Sigma DD$ itself and its implementation. After what we analyze benchmarks. Then we compare our work to other approaches and particularly to the more traditional Term Graph Rewriting. Finally, we conclude and discuss the open issues and future work.

2 Order-Sorted Term Rewriting Systems

Here we recall some usual definitions required to define Order-Sorted Term Rewriting of Ordered Algebraic Specification formally. For more details see [9].

Definition 2.1 [Sort & S-Sorted Set] Let $S$ be a finite set of sorts. A $S$-sorted set $A$ is a union of a family of sets indexed by $S$ ($A = \bigcup_{s \in S} A_s$), noted as $A = (A_s)_{s \in S}$. Informally an S-Sorted Set is a partitioned set, not always disjoint, in which the partitions are determined by the sorts names.

A Signature defines the names of the operations among the sorts. This enables composition of said operations to build terms.

Definition 2.2 [Order-Sorted Signature and Terms] Let $\leq \subseteq (S \times S)$ be a partial order. We extend the ordering $\leq$ on $S$ to words of equal length of $S^*$ by $s_1, \ldots, s_n$ iff $\forall i \ s_i \leq s'_i$ with $1 \leq i \leq n$. Similarly we extend $\leq$ to pairs $S^* \times S$ by $(w, s) \leq (w', s')$ iff $w \leq w'$ and $s \leq s'$. An order-sorted signature is a triple $\Sigma = (S, \leq, F)$, where $S$ is a finite set of sorts, $\langle S, \leq \rangle$ is a partially ordered set of sorts and $F = (F_{w,s})_{w\in S^*, s\in S}$ is a $(S^* \times S)$-sorted set of function names.

We often denote a function name $f \in F_{s_1,\ldots,s_n,s}$ by $f : s_1, \ldots, s_n \rightarrow s$ and $f : \rightarrow s$ if $f \in F_{\epsilon,s}$ where $\epsilon$ denotes the empty word.

The set of terms of $\Sigma$ over $X$ is a $S$-sorted set $T_{\Sigma,X}$, where each set $(T_{\Sigma,X})_s$ is the least set inductively defined by:

- $x \in (T_{\Sigma,X})_s$, $\forall x \in X_s$.
- $f \in (T_{\Sigma,X})_s$, $\forall f : \rightarrow s'$ such that $s' \leq s$ and is called a constant.
- for all operations that are not a constant: $f(t_1, \ldots, t_n) \in (T_{\Sigma,X})_s$, $\forall f : s_1, \ldots, s_n \rightarrow s'$ such that $s' \leq s$ and $\forall t_i \in (T_{\Sigma,X})_{s_i}$ with $1 \leq i \leq n$.

We also define $\tau : (T_{\Sigma,X})_s \rightarrow S$, the typing function s.t $\forall t \in (T_{\Sigma,X})_s$, $\tau(t) = s$.

Definition 2.3 [Context and Sub-terms] Let $\Sigma = (S, \leq, F)$ be an order-sorted signature and $X$ be a $S$-sorted variable set, let also $\square \notin F \cup X$ be a special constant symbol called a placeholder. A context $C$ of a term $t \in T_{\Sigma,X}$ is a term $(T_{\Sigma,(\square)\cup X})_s$.
such that if $C_t[\square_1, \ldots, \square_n]$ is a context with $n$ occurrences of $\square$ and $t_1, \ldots, t_n$ are terms $\in (T_{S, \leq}(\square), X)_s$, then $C_t[t_1, \ldots, t_n]$ is the result of replacing the $\square_i$ by the $t_i$.

A term $st \in (T_{S, \leq}(X))_s$ is a sub-term of $t \in (T_{S, \leq}(X))_s$ noted $st \subseteq t$ if there exists a context $C$ of term $t$ denoted $C_t[\_]$ such that $t = C_t[st]$.

**Definition 2.4** [Substitution] Let $\Sigma = \langle S, \leq, F \rangle$ be an order-sorted signature and $X$ be a $S$-sorted variable set. A substitution $\sigma$ is a family of mappings $\forall s \in S, \sigma : X_s \rightarrow (T_{S, \leq}(X))_{s'}$, where $s' \in S$ and $s' \leq s$. Every substitution $\sigma$ extends uniquely to a family of morphisms $\forall s \in S, \sigma^\# : (T_{S, \leq}(X))_s \rightarrow (T_{S, \leq}(X))_{s'}$, where $s, s' \in S$ and $s' \leq s$:

- $\sigma^\#(f(t_1, \ldots, t_n)) = f(\sigma^\#(t_1), \ldots, \sigma^\#(t_n))$
- $\sigma^\#(f_s) = f_s$ with $f_s \in F_{s, s}$
- $\sigma^\#(x_s) = \sigma(x_s)$

**Definition 2.5** [Conditional Rewrite Rule] Let $\Sigma = \langle S, \leq, F \rangle$ be an order-sorted signature and $X$ be a $S$-sorted set of variables. A rewrite rule is a tuple $(l, r, con)$ with $\exists s \in S, l, r \in (T_{S, \leq}(X))_s$ and $con$ a conjunction of equalities between terms of $T_{S, \leq}(X)$ s.t. $l \not\in X$, $\text{var}(r) \subseteq \text{var}(l)$ and $\text{var}(\text{cond}) \subseteq \text{var}(l)$. A rewrite rule is noted $l \sim_{\text{cond}} r$ or simply $l \sim r$ if there is no condition. We note $\text{Rew}_{S, \leq}(X)$ a set of rewrite rules w.r.t $\Sigma$ and $X$.

**Definition 2.6** [Rewriting step] Let $\Sigma = \langle S, \leq, F \rangle$ be an order-sorted signature, $X$ be a $S$-sorted set of variables and $\text{rule} : l \sim_{\text{cond}} r$ with $l, r \in T_{S, \leq}(X)$ a conditional rewrite rule. Let also $t$ and $t' \in T_{S, \leq}(X)$ be two terms. The pair $(t, t')$ is called a rewriting step if there exists a context $C_t$ of term $t$ and a substitution $\sigma$ such that: $t = C_t[\sigma^\#(l)]$ and $t' = C_t[\sigma^\#(r)]$. We note $t \sim_{\text{rule}} t'$ this rewriting step. Besides, $\sim$ is extended to all rules and noted $\sim_{\text{Rew}_{S, \leq}(X)}$.

Let a term $t \in T_{S, \leq}(X)$, $t$ is said to be in normal form (irreductible) iff:

$$\nexists t' \in T_{S, \leq}(X) \text{ such that } t \sim_{\text{Rew}_{S, \leq}(X)} t'.$$

We will use the Innermost Rewriting strategy in the sequel. It is a bottom-up algorithm that proceeds by first normalizing the sub-terms of a term. When all sub-terms are reduced to a normal form, the term itself is considered for reduction.

### 3 Encoding formalism

Data Decision Diagrams (DDD) and Set Decision Diagrams (SDD) are both evolutions of the well-known Binary Decision Diagrams (BDD) [7]. While BDD is often seen as representing a Boolean function, it can also be seen as a set of sequences of assignments of Boolean values to variables. DDD (resp. SDD) are similar but for any kind of values (resp. sets) of the form $(\text{var}_1 := \text{val}_1), (\text{var}_2 := \text{val}_2) \ldots (\text{var}_n := \text{val}_n)$. In the sequel, $E$ is the set of variable and $\forall e \in E, \text{Dom}(e)$ is the set of values that can be taken by the variable $e$.

0 represents the empty Decision Diagrams, a sequence that finishes with 0 does not exist (like in ZBDD), 1 represents an existing sequence of assignments. 0 and 1 are also called terminals because they are the last node of any sequences.

To handle complex structures, being able to assign single values (BDD or DDD) to variables is not enough. Set Decision Diagrams (SDD) solve that problem by
The SDD on the left side represents the Cartesian product of $x$ and $y$ that is 9 paths or states. The SDD $(e_{1\text{SDD}} = x, y)$ embeds another SDD $(e_{2\text{SDD}} = \{a, b\})$:

$$
\begin{align*}
&x \xrightarrow{a (1)} y \xrightarrow{b (1)} 1 + x \xrightarrow{a (0)} y \xrightarrow{b (2)} 1 + \\
&x \xrightarrow{a (2)} y \xrightarrow{b (0)} 1 + x \xrightarrow{a (0)} y \xrightarrow{b (2)} 1 + \\
&x \xrightarrow{a (1)} y \xrightarrow{b (1)} 1 + x \xrightarrow{a (1)} y \xrightarrow{b (1)} 1 + \\
&x \xrightarrow{a (2)} y \xrightarrow{b (0)} 1 + x \xrightarrow{a (2)} y \xrightarrow{b (0)} 1 +
\end{align*}
$$

The power of the SDD lies in the symbolic encoding of the Cartesian product. Using SDD, thanks to the sets, we end up with a two-dimensional symbolic encoding.

Fig. 1. Example of a SDD

allowing assignments to be sets. Arcs of the SDD represent a set instead of a boolean value. We assume no variable ordering, and the same variable can occur several times in an assignment sequence. In the following, $E$ denotes a set of variables, and for any $e \in E$, $Dom(e)$ represents the domain of $e$ which may be infinite.

**Definition 3.1** [Set Decision Diagrams] The SDD set $S$ is the least set such that:

- $\{0, 1\} \subseteq S$
- $(e, a) \in S$ with:
  - $e \in E$ with $E$ the set of all SDD variables.
  - $\alpha : \pi \rightarrow S$, with $\pi = \{a_0, \ldots, a_i, \ldots, a_n\}$ a disjoint partition of $Dom(e)$ s.t.
    $\forall a_i, a_j \in \pi$, with $i \neq j$, $\alpha(a_i) \neq \alpha(a_j)$.

Def. 3.1 ensures that any set of assignment sequences has a unique (canonical) SDD representation. Indeed $\pi$ is a disjoint partition and no two arcs from a node lead to the same sub-graph. $e \xrightarrow{d} d$ denotes the $SDD(e, \alpha)$ with $\alpha(x) = d$. Fig. 3 gives an example of a SDD that embeds another SDD. Solid arrows are pointers to sub-SDD while dashed ones are pointers to values that can be themselves SDD.

Since SDD represent sets, we can define the usual set operations on them such as $\cup_{\text{SDD}}, \cap_{\text{SDD}}, \setminus_{\text{SDD}}$ or $\cup, \cap, \setminus$ if there is no possible confusion. For a definition of the set operations on SDD, see [5,12]. Concatenation $d_1 \otimes_{\text{SDD}} d_2$ concatenates $d_2$ to every terminal of $d_1$. This concatenation corresponds to the Cartesian product of the sets represented by both sdd. Operations from set theory as well as homomorphism application (see Def. 3.3) should be performed on compatible SDD. This is the counterpart of BDD ordering.

**Definition 3.2** [SDD compatibility] Two SDD are said compatible iff their sequences are compatible. Two sequences $s = e_1 \xrightarrow{x_1} \ldots 1$ and $s' = e'_1 \xrightarrow{x'_1} \ldots 1$ are compatible (noted $s \approx s'$) iff:

- $s = s' = 1$
- $s = e \xrightarrow{d} d \land s' = e' \xrightarrow{d'} d'$ such that $e = e'$ and
  - $x \approx x'$ if $x, x' \in S$ (other referenced type should define their own compatibility)
\[ d \approx d' \text{ if } x \cap x' \neq \emptyset \]

Unlike work on binary decision diagrams, operators are not limited to those predefined. Indeed one of the strengths of the SDD structures is their support of so-called inductive homomorphisms. Namely, operations that are inductively defined on the structure of the SDD and that are compatible with the \textit{union} operator. This compatibility induces a high efficiency of user defined operations. A homomorphism is a mapping \( \phi \) from \( S \) to itself s.t \( \phi(0) = 0 \) and \( \phi(d \cup d') = \phi(d) \cup \phi(d') \), \( \forall d, d' \in S \).

The \textit{union} (\( \cup \)) and the composition (\( \circ \)) of two homomorphisms are homomorphisms. Since a decision diagram is inductively defined, operations on them can also be inductively defined. This allows the user to give a local definition of the homomorphism i.e. what it should do with a given pair \( \langle \text{variable}, \text{value} \rangle \).

**Definition 3.3** [Inductive Homomorphisms on SDD] Let \( \phi_{e,x} \) with \( e \in E \) and \( x \in \text{Dom}(e) \) be a family of homomorphisms and \( d_1 \) a SDD:

\[
\forall d \in S, \phi(d) = \begin{cases} 
0 & \text{if } d = 0 \\
\; d_1 & \text{if } d = 1 \\
\bigcup_{x \in \text{Dom}(e)} \phi_{e,x}(e \xrightarrow{x} \alpha(x)) & \text{if } d = \langle e, \alpha \rangle
\end{cases}
\]

is an inductive homomorphism. 
\( \phi^*(d) \) represents the fixpoint application of \( \phi \) on \( d \). That is when \( \phi^* = \phi^n \) with \( n \) the smallest integer such that \( \phi^n(d) = \phi^{n-1}(d) \).

Example: Let suppose we want to define a function \( \phi_{\text{add}_1} \) that adds the set \( v_1 \) to every non-empty value and returns 1 when reaching the terminal.

\[
\phi_{\text{add}_1}(e \xrightarrow{x} d) = \begin{cases} 
eq 0 \rightarrow \phi_{\text{add}_1}(d) & \text{if } x \neq 0 \\
\phi_{\text{add}_1}(d) & \text{otherwise}
\end{cases}, \phi_{\text{add}_1}(1) = 1
\]

As for the set operations, inductive homomorphism can be evaluated lazily saving both memory and processing time.

Please note that once created, a DD is never altered. When a set operation or a homomorphism is applied to a DD, a new DD is created. This DD is checked for existence in a uniqueness table. Either the DD has already been registered and the reference is returned or the new reference is added to the uniqueness table.

From an implementation point of view, we can leverage on the canonicity (thanks to the DD creation and DD union operator) of the representation in order to implement constant time equality between DD and thus implement efficient caching.

Set operations or homomorphisms are applied on an inductive structure and thus each processing step can be put in the cache for further use. This is very useful to save computing time and it allows to implement efficiently fix-point computations.

### 4 \( \Sigma DD \)

In this section, we define the \( \Sigma DD \) and operations on them formally. Since the goal is to rewrite sets of terms, we need the standard operations from the set theory (union, intersection and difference) and a way to represent such sets efficiently.
4.1 Term Encoding and Extraction

Definition 4.1 [\(\Sigma DD\)] Let \(\Sigma = \langle S, \leq, F \rangle\) be an order-sorted signature, \(X\) be an S-sorted set of variables and \(T_{\Sigma,X}\) be an S-sorted set of terms. The \(S\)-indexed set \(\Sigma_{DD}\) over signature \(\Sigma\) is inductively defined as the least set \(s.t:\)
- \(\{0, 1\} \subseteq (\Sigma_{DD})_s\) with 0, the empty \(\Sigma DD\) and thus the empty set of terms.
- \(\langle s, \alpha \rangle \in (\Sigma_{DD})_s\) with:
  - \(s \in S\) which represents the sort of the head sub-\(\Sigma DD\).
  - \(\alpha : \pi \rightarrow (\Sigma_{DD})_s\), with \(\pi = \{a_1, \ldots, a_n\}\) a disjoint partition of \((\Sigma_{DD})_s \cup F_s \cup X_s\) \(s.t.\) \(\forall a_i, a_j \in \pi, with 1 \leq i, j \leq n\) and \(i \neq j, \alpha(a_i) \neq \alpha(a_j)\).
  - Moreover \(\forall a_i \in \pi, a_i \in (\Sigma_{DD})_{s'} with s' \leq s\).

Not all \(\Sigma DD\) obtained from the previous definition are well-formed. For example 1 is not a well-formed \(\Sigma DD\) since it does not have any semantic as a term (it does not represent a valid term).

Definition 4.2 [Well-formed \(\Sigma DD\)] Well formed \(\Sigma DD\) are those of the form:
- 0 which is the empty set of terms.
- \(s \xrightarrow{a} 1\) with \(a \subseteq F_{e,s} \cup X_s\) a set of constants and variables.
- \(s \xrightarrow{op} w_1 \xrightarrow{v_1} 1\) with \(op \subseteq F_{w_1,s}\), \(|w_1| = 1\) for unary operators (such as succ).
- \(s \xrightarrow{op} w_1 \xrightarrow{v_1} \ldots w_n \xrightarrow{v_n} 1\) with \(op \subseteq F_{w_1 \ldots w_n,s}\) and \(v_1, \ldots, v_n \in (\Sigma_{DD})_{w_i}, 1 \leq i \leq n\) the set of composed terms \((n \geq 2)\).

Example: The following \(\Sigma DD\) : \(\mathbb{B} \geq \mathbb{Z} \xrightarrow{N-1} \mathbb{Z} \xrightarrow{N + 1} \mathbb{N} \xrightarrow{N \cdot \text{suc} \cdot N-1} \mathbb{N} \xrightarrow{N-1} 1\) is the encoding of term \(> (0, +(0, \text{suc}(0)))\) with \(S = \{\mathbb{B}, \mathbb{N}, \mathbb{Z}\}\), \(F = \{> : \mathbb{Z}, \mathbb{Z} \rightarrow \mathbb{B}, + : \mathbb{N}, \mathbb{N} \rightarrow \mathbb{N}, \text{suc} : \mathbb{N} \rightarrow \mathbb{N}, 0 : \rightarrow \mathbb{N}\}\) and \(\leq = \{\mathbb{N} < \mathbb{Z}\}\).

As previously mentioned, union is an important operation to ensure the canonicity of the representation. This canonicity is very important to enable constant time equality and efficient caching. \(\Sigma DD\) union is extended from the \(SDD\) union as defined in [12]. The main difference is the support of order-sorting and hence the notion of compatible variable.

Definition 4.3 [Compatible \(\Sigma DD\) & Union] As for \(SDD\) we have the notion of compatible \(\Sigma DD\). Two \(\Sigma DD\) \(s \xrightarrow{x} v\) and \(s' \xrightarrow{x'} v'\) with \(v, v' \in \Sigma_{DD}\) are compat-
ble iff:

- $s$ and $s'$ are compatible iff $s \leq s'$ or $s' \leq s$.
- $s \xrightarrow{x} v$ and $s' \xrightarrow{x'} v'$ are compatible, iff $s$ and $s'$ are compatible, $v$ and $v'$ are compatible and finally $x$ and $x'$ are compatible.

This extended compatibility enables union between $\Sigma DD$ with two sorts that are in the same hierarchy, the union always returns the less specific one:

\[
 s \xrightarrow{\{x\}} 1 \cup s' \xrightarrow{\{x'\}} 1 = s' \xrightarrow{\{x \cup x'\}} 1 \text{ if } s \leq s' \text{ and } s \xrightarrow{\{x \cup x'\}} 1 \text{ otherwise.}
\]

Example: With $\mathbb{N} \leq \mathbb{Z}$, following union $\mathbb{N} \xrightarrow{\{1\}} 1 \cup \mathbb{Z} \xrightarrow{\{-1,1\}} 1$ returns $\mathbb{Z} \xrightarrow{\{-1,1\}} 1$.

The rest of the union operation as well as the other set operations remain identical to the SDD set operations as defined in [12].

We will now analyze how to encode a set of terms as a well-formed $\Sigma DD$. Since we extended the notion of compatible Decision Diagrams, it enables us to handle order sorting.

**Definition 4.4 $\Sigma DD$ encoding homomorphism**

Let $\Sigma = \langle S, \leq, F \rangle$ be a signature and $X$ be an $S$-sorted set of variables. Let also $T$ be a set of terms, $T \subseteq T_{\Sigma,X}$ and $t \in T_{\Sigma,X}$. The encoding morphism $en : \mathcal{P}(T_{\Sigma,X}) \rightarrow \Sigma G D D_{\Sigma}$ is inductively defined:

- $en(\emptyset) = 0$
- $en(\{t\}) = s \xrightarrow{\{x\}} 1$, if $t = x$ with $x \in X_s$
- $en(\{t\}) = s \xrightarrow{\{k\}} 1$, if $t = k$ with $k \in F_{e,s}$
- $en(\{t\}) = s \xrightarrow{\{f\}} s_1 \xrightarrow{en(\{t_1\})} \ldots s_n \xrightarrow{en(\{t_n\})} 1$, if $t = f(t_1, \ldots, t_n)$ with $f : s_1, \ldots, s_n \rightarrow s$
- $en(\{t\} \cup T) = en(\{t\}) \cup en(T)$ where the union between $\Sigma DD$ is defined in definition 4.3.

Sets are first class citizens in this definition. This enables easy encoding of sets of terms, which is a very important feature is number of applications.

**Fig. 2** shows the difference in sharing between regular term graph encoding (left side) and $\Sigma DD$ encoding (right side). Both graphs represent the set of terms: $\{+(suc(0), suc(suc(0))), + (suc(0), suc(0))\}$.

In the $\Sigma DD$ approach, the second operand is treated as one sub graph and thus both terms can be rewritten in one sequence.

**Fig. 2. $\Sigma DD$: Term Graph vs $\Sigma DD$**

Fig. 2 shows a $\Sigma DD$. The arrows represent pointers to other $\Sigma DD$. While solid arrows represent pointers to sub-$\Sigma DD$, the dashed ones represent pointers to values that can be themselves $\Sigma DD$. Terminals (1) are duplicated for readability reasons.
We need to be able to extract the set of terms that are encoded in a $\Sigma DD$. The extracting morphism takes a well-formed $\Sigma DD$ and returns its associated set of terms.

**Definition 4.5** [$\Sigma DD$ extracting morphism] Let $\Sigma = (S, \leq, F)$ be a signature and $X$ be an $S$-sorted set of variables. Let also $T$ be a set of terms over $\langle \Sigma, X \rangle$ noted $T_{\Sigma,X}$. The extracting morphism $ex : \Sigma GDD_\Sigma \rightarrow \mathcal{P}(T_{\Sigma,X})$ is inductively defined by:

- $ex(0) = \emptyset$, the empty set of terms.
- $ex(\nu_T) = T_s$, if $\nu_T = s \xrightarrow{T_s} 1$ with $T_s \subseteq X_s \cup F_{\epsilon,s}$
- $ex(\nu_T) = \bigcup_{f \in T_s} \bigcup_{s_1,...,s_n} \prod_{i=1}^{n} ex(v_{ST_i}) f(s_1,\ldots,s_n)$, if $\nu_T = s \xrightarrow{T_s} s_1 \xrightarrow{\nu_{ST_1}} \ldots s_n \xrightarrow{\nu_{ST_n}} 1$ with $T_s \subseteq F_{w,s}$, $w = (s_1,s_2,...,s_n) \neq \epsilon$ and $\prod_{i=1}^{n} ex(v_{ST_i}) = \cup s_1 \in ex(v_{ST_1}) \ldots \cup s_n \in ex(v_{ST_n}) \langle s_1,\ldots,s_n \rangle$

This complex union shows the power of $\Sigma DD$-encoding w.r.t the encoding of sets of terms.

**Lemma 4.6 (ex is a homomorphism w.r.t union)** $\forall \nu_T, \nu_{T'} \in \Sigma GDD$ we have $ex(\nu_T \cup \nu_{T'}) = ex(\nu_T) \cup ex(\nu_{T'})$ Proof by induction on the structure of the $\Sigma DD$.

In order for encoded/extracted terms to remain consistent, we must prove that whenever we extract a previously encoded set of terms, it remains identical.

**Lemma 4.7 (Identity morphism on $\mathcal{P}(T_{\Sigma,X})$)** By structural induction we show that $\forall T_s \subseteq T_{\Sigma,X}$, $ex(en(T_s)) = T_s$.

Let’s establish the base case:

- If $T_s \subseteq X_s \cup F_{\epsilon,s}$, $en(T_s) = s \xrightarrow{T_s} 1$ and $ex(s \xrightarrow{T_s} 1) = T_s$ straightforward by Def. 4.4 and Def. 4.5

Now for the inductive step, consider the set of terms $T_s = \{t_s\} \cup T'_s$ with $t_s = f(t_1\ldots t_n)$ where $f \in F_{w,s}$ and $t_1\ldots t_n \in T_{\Sigma,X}$:

- $ex(en(\{t_s\})) = t_s$ by Def. 4.4 and Def. 4.5 and
- $ex(en(T_s)) = ex(en(\{t_s\}) \cup T'_s)$
- $ex(en(\{t_s\} \cup T'_s)) = ex(en(\{t_s\})) \cup ex(en(T'_s))$ because $en$ and $ex$ are homomorphisms.

Therefore, $ex \circ en$ is the identity morphism on $\mathcal{P}(T_{\Sigma,X})$.

To prove the bijection, we first prove that $en \circ ex$ is the identity morphism and thus that $en$ and $ex$ are isomorphisms.

**Lemma 4.8 (identity morphism on $\Sigma GDD_\Sigma$)** As for lemma 4.7, by structural induction, we show that $\forall v \in \Sigma GDD$ we have $en(ex(v)) = v$. Thus, we deduce that $en \circ ex$ is the identity morphism on $\mathcal{P}(T_{\Sigma,X})$.

Then by the fact that $en$ and $ex$ are homomorphisms they are isomorphic, so the representation by $\Sigma DD$ is consistent.

**Theorem 4.9 (Canonicity)** Since $en$ and $ex$ are isomorphisms and consequently bijections they guarantee the canonicity.
\[\forall t_1, t_2 \in T_{X, \Sigma},\ en(\{t_1\}) = en(\{t_2\}) \iff t_1 = t_2.\]

\[\forall v_1, v_2 \in \SigmaDD_{\Sigma},\ ex(v_1) = ex(v_2) \iff v_1 = v_2.\]

It is worth mentioning that variable ordering is very important to the \(\Sigma\)DD efficiency. We took a straightforward variable ordering (based on operators’ arity because finding an optimal variable ordering is a NP-complete problem. Usually, Decision Diagrams rely on heuristics and ad-hoc solutions to find an efficient ordering. We plan to work on this in a near future.

4.2 Rewriting operations on \(\Sigma\)DD

To reach a normal form, one should choose a rewriting strategy. We have implemented both innermost and outermost rewriting. In the sequel, we will only present innermost rewriting. The reader should be able to adapt easily the definitions for outermost rewriting.

Please note that a \(\Sigma\)DD is said to be in a canonical form (normal form) as soon as the application of the \(\phi_{InrMstRw}\) homomorphism reaches a fixpoint. Alternatively, since we do not support axioms between generators, if it is by construction only built upon generators.

**Definition 4.10** [InnerMost Rewriting Homomorphism]

Let \(t \in \SigmaDD_{\Sigma}\) be a \(\Sigma\)DD (set of terms) to rewrite and \(\phi_{InrMstRw}\) the homomorphism that rewrites a \(\Sigma\)DD until it is reduced to a canonical form. Please note that \(\phi_{InrMstRw}\) stands for the fixpoint application of the \(\phi_{InrMstRw}\) homomorphism as described in Def. 3.3.

\[\phi_{InrMstRw}(s \xrightarrow{x} v) = \begin{cases} s \phi_{InrMstRw}(x) & \text{if } x \text{ and } v \in \SigmaDD_{\Sigma} \\ \phi_{Apply\,s \leftarrow \SigmaDD}(s \xrightarrow{x} \phi_{InrMstRw}(v)) & \text{if } v \in \SigmaDD_{\Sigma} \text{ and } x \subseteq F \end{cases}\]

\(\phi_{InrMstRw}(1) = 1\)

\(\phi_{InrMstRw}\) first checks whether the current value on the arc is a \(\Sigma\)DD or a set of operators. In the first case, it reduces the \(\Sigma\)DD to a canonical form and does the same for the sub-graph. In the second case, it applies the \(\phi_{Apply}\) homomorphism to the whole graph after having reduced the sub-graph. The \(\phi_{Apply}\) homomorphism applies a reduction rule on the given \(\Sigma\)DD and thus performs a rewriting step. As previously mentioned, homomorphism applications can be cached, it is not mandatory. If we want to simulate term rewriting, we need to avoid caching of rule application in order to get a different result when rewriting the same sub-term.

4.2.1 Rewriting Step

Given a rewriting rule \(\text{Rule} = l \leadsto r\), performing graph rewriting is usually expressed using the following equation: \(G_T' = (G_T \setminus G_l) \cup G_r\). In which \(G_T'\) represents the result of the transformation, \(G_T\) is the host graph that is the graph on which the transformation is applied, \(G_l\) (resp. \(G_r\)) the left(resp. right) graph namely the graph that matches the left (resp. right) term of the rewriting rule. Obviously since \(G_T\) is, in our case, a \(\Sigma\)DD, rewriting operations are applied on a set of terms.

The \(\phi_{Apply}\) homomorphism, applies a rewriting step on a well-formed \(\Sigma\)DD as explained in Def. 2.6. By extending \(\sigma^\#\) to \(\Sigma\)DD we have \(G_l = \sigma^\#(en(l))\),
\[ G_r = \sigma^\#(en(r)). \]

**Definition 4.11** [Rule Application Homomorphism] Let \( op \in F \) be an operation symbol and let \( \phi_{pm,op} \) the pattern matcher homomorphism that unifies a \( \Sigma DD \) with the left-hand side of rewriting rules starting with operating symbol \( op \) and returns the pair \( (G_l, G_r) \). The rule application homomorphism is defined by:

\[
\phi_{Apply_{\Sigma DD}}(s \xrightarrow{x} v) = \begin{cases} 
  1 & \text{if } v = 1 \text{ and } x \subseteq F_{e,s} \cup X_s \\
  \bigcup_{op \in x}(G_T \setminus G_l) \cup G_r & \text{with } G_T = s \xrightarrow{x} v \text{ and } \\
  \langle G_l, G_r \rangle = \phi_{pm,rule_{\Sigma DD}}(G_T) & \text{otherwise.}
\end{cases}
\]

\[
\phi_{Apply_{\Sigma DD}}(1) = 1
\]

To apply a rule we must first check whether a given DD fulfills a given pattern and thus does qualify for a given rewriting rule. Moreover, the pattern matcher needs to create a substitution that can later be used to check the conditions of application of the rule and build the left and right-hand side. The pattern matchers are built from the axioms. If several axioms share the same left-hand side, they must have different conditions of application.

**Definition 4.12** [Pattern Matcher Homomorphism] Let \( Spec = \langle \Sigma, X, E \rangle \) be an ordered algebraic specification, let \( Rew_{Spec} \) be the set of rewriting rules built upon \( Spec \), \( rule = (l, r, cond) \in Rew_{Spec} \) and its encoding \( rule_{\Sigma DD} = \langle en(l), en(r), en(cond) \rangle = \langle v_l, v_r, v_{cond} \rangle \) be a rewriting rule with \( v_l = s \xrightarrow{op} v' \). Let \( rule_{\Sigma DD}' = \langle v', v_r, v_{cond} \rangle \) be the sub-term to match. The rule-sorted set of pattern matcher homomorphism \( (\phi_{pm})_{rule \in Rew_{Spec}} \) is defined by:

\[
\phi_{pm,rule_{\Sigma DD},\sigma}(s \xrightarrow{op} v) = \begin{cases} 
  \phi_{pm,rule_{\Sigma DD},\sigma}(v), \text{ if } v_l = s \xrightarrow{\{x\}} v', \text{ and } x \in X_s \\
  \phi_{pm,rule_{\Sigma DD},\sigma}(v), \text{ if } v_l = s \xrightarrow{\{f\}} v', \text{ if } f \in F_{e,s} \text{ and } f \in op \\
  \phi_{pm,rule_{\Sigma DD},\sigma}(op) \circ \phi_{pm,rule_{\Sigma DD},\sigma}(v), \text{ if } v_l = s \xrightarrow{op'} v' \\
  0, \text{ otherwise}
\end{cases}
\]

where \( op' \in SIGDD \) and \( rule''_{\Sigma DD} = \langle op', 0, 0 \rangle \).

\( \sigma \cup \langle x, op \rangle \) stands for adding a pair \( \langle \text{variable, value} \rangle \) to the current substitution \( \sigma \).

If the inductive homomorphism gets to the terminal node (1), at least, one of the \( \Sigma DD \) fulfills the \( l \) pattern. In that case, the homomorphism has to check whether the conditions of application has also been fulfilled. If so, it returns both \( l \) and \( r \) \footnote{An inductive homomorphism can only return a \( \Sigma DD \) and, therefore, we have to wrap up the left-hand side and the right and side in a single \( \Sigma DD \). To do this we add two dummy sorts \( l \) and \( r \) to the set of sorts \( S \) such that we can create a \( \Sigma DD \) of the form \( l \xrightarrow{G_l} r \xrightarrow{G_r} 1 \) in the \( \phi_{pm,rule_{\Sigma DD},\sigma}(1) \).} \footnote{An inductive homomorphism can only return a \( \Sigma DD \) and, therefore, we have to wrap up the left-hand side and the right and side in a single \( \Sigma DD \). To do this we add two dummy sorts \( l \) and \( r \) to the set of sorts \( S \) such that we can create a \( \Sigma DD \) of the form \( l \xrightarrow{G_l} r \xrightarrow{G_r} 1 \) in the \( \phi_{pm,rule_{\Sigma DD},\sigma}(1) \).}. If \( r \) is empty (0), it means we are currently in a sub-term matcher (not at the top level). In this case, \( \phi_{pm,rule_{\Sigma DD},\sigma} \) simply returns 1 because the conditions can only be checked when the complete term has been scanned.
The last homomorphism substitutes values to variables. Namely, it walks through the graph and each time it crosses a variable it tries to replace it. The extending substitution $\sigma^+$ (see Def. 2.4) is embedded as a parameter.

Definition 4.13 [Substitution Homomorphism] Let $Spec = \langle \Sigma, X, E \rangle$ be an ordered algebraic specification and $\sigma$ a substitution. The substitution homomorphism $\phi_{subst,\sigma}$ is defined by:

$$\phi_{subst,\sigma}(t) = \begin{cases} 1 & \text{if } \nu_r = 0 \\ 1 \quad & \text{if } \phi_{subst,\sigma}(\nu_l) \Rightarrow_{\phi_{subst,\sigma}(\nu_r)} \phi_{subst,\sigma}(\nu_l) \equiv \langle G_l, G_r \rangle \text{ of } \phi_{Apply} \\ \phi_{subst,\sigma}(t) & \text{otherwise} \\ 0 & \text{otherwise} \end{cases}$$

Theorem 4.14 (Rewriting-step equivalence of a single term in $\Sigma DD$)
Application of the $\phi_{Apply}$ homomorphism is equivalent to a rewriting step:

$$\forall t, t' \in T_{\Sigma}; \quad t \stackrel{\text{rule}}{\Rightarrow} t' \iff \phi_{Apply_{rule_{\Sigma}}}(en(\{t\})) = en(\{t'\})$$

Proof: By induction on the terms and the definition of the $\phi_{Apply}$ homomorphism which is, by construction, similar to a rewriting step.

Corollary 4.15 (Rewriting-step equivalence of a set of terms in $\Sigma DD$)

$$\forall T, T' \subseteq T_{\Sigma}; \quad T \stackrel{\text{rule}}{\Rightarrow} T' \iff \phi_{Apply_{rule_{\Sigma}}}(en(T)) = en(T')$$

because $en$ is an isomorphism. Please note that the correspondence between each element of $T$ and its image in $T'$ is lost.

Corollary 4.16 ($\phi_{InrMstRw}$ preserves the termination and confluence)
Under termination and confluence hypothesis of the rewrite system, we provide by rewriting on $\Sigma DD$ values, a valid and complete calculus s.t.: $\forall t, t' \in T_{\Sigma}$;
\[ \text{Rew}^*(t) = \text{Rew}^*(t') \Leftrightarrow \phi_{\text{InrMstRw}}^*(\text{en}(\{t\})) = \phi_{\text{InrMstRw}}^*(\text{en}(\{t'\})). \]

The terms: \((\text{suc}(0), \text{suc}(0))\) and \((\text{suc}(\text{suc}(0)), \text{suc}(0))\) are rewritten using the following rewriting rules:

- \((\text{suc}(x), y) \leadsto \text{suc}(+(x, y))\)
- \((0, x) \leadsto x\)

For the sake of simplicity, some pointers and parts of the graph (i.e. values such as +, suc, 0 and \(\mathbb{N} \xrightarrow{\text{suc}} \mathbb{N} \xrightarrow{0} 1\)) are drawn several times although they are shared in the implementation thanks to the canonicity.

**5 Implementation & Benchmarks**

### 5.1 Implementation

The \(\Sigma DD\) library is implemented in Java and requires at least version 5 since it heavily uses the new language features such as generics, variable arguments size or boxing. The library is built on top of the JDD [13] library that provides support for the Data Decision Diagrams and the Set Decision Diagrams in Java. Both libraries are freely available under the GNU license at [http://smv.unige.ch/tiki-list_file_gallery.php?galleryId=59](http://smv.unige.ch/tiki-list_file_gallery.php?galleryId=59). Even if Java provides an efficient object creation, performant garbage collection and a very good tooling, the performances of the implementation suffer from the lack of tail recursion. This issue may be solved in a future version of Java. The \(\Sigma DD\) library provides a user friendly API to describe an AADT, build terms and perform rewriting.

Both libraries (JDD & \(\Sigma DD\)) have been successfully used in our model checker called AlPiNA [11] that enables reachability analysis on Algebraic Petri Net.

### 5.2 Benchmarks

Although we have performed benchmarks\(^3\) on our implementation, they are not exhaustive and must be improved. We present here some results to compare \(\Sigma DD\)'s performances to the well-known rewriter Maude [14]. We used a new feature of the version 2.4 namely built-in support for sets. The following figures are given as an indication to illustrate our approach. The benchmarks are based on the Algebraic Specification of naturals because it is well known and easy to understand.

\(^3\) Performed on a Mac Book Pro with 1 Intel Core 2 Duo at 2.5 Ghz and 2GB of RAM.
The first benchmark rewrites a term and indicates that Maude is 35 times faster than \(\Sigma DD\) when rewriting a single term with poor or no sub-term sharing. In this case, both rewriters used the same number of steps (10002). This indicates that the cost of one rewriting step is 35 times higher using \(\Sigma DD\). One reason is the management of sets even for single term rewriting. The second one illustrates the impact of the cache in a single term rewriting, thanks to this, the number of rewrites is much smaller in the \(\Sigma DD\) case. The third example uses a conditional rewriting rule: \(-((suc(x), suc(y)) \sim (x, y))\) if \(y < x\). Thanks to the cache, \(\Sigma DD\) is better (10x) than Maude due to the inductive definition of this axiom, and thus the high caching. The fourth example proves a property on a set of terms, namely that for all \(i\) s.t. \(0 \leq i \leq 80\), \(\sum_{x=0}^{i} x \leq i^2\). Sharing and caching help to reduce the difference (2x). When performing model checking on Algebraic Petri Nets, we must often rewrite sets of very similar terms. This is a very favorable case for \(\Sigma DD\) because each rewriting step is applied to the whole set. In the last case, \(\Sigma DD\) is 42x faster than Maude.

<table>
<thead>
<tr>
<th>Terms</th>
<th>(\Sigma DD)</th>
<th>Maude</th>
</tr>
</thead>
<tbody>
<tr>
<td>((1000 + 5000) + 5000)</td>
<td>1</td>
<td>0.028</td>
</tr>
<tr>
<td>Single term (sub-term sharing)</td>
<td>6.99</td>
<td>0.031</td>
</tr>
<tr>
<td>((1000 + 5000) - 5000) (conditions)</td>
<td>834</td>
<td>0.74</td>
</tr>
<tr>
<td>(\forall i \leq 80, \sum_{x=0}^{i} x \leq i^2)</td>
<td>39.54</td>
<td>0.47</td>
</tr>
<tr>
<td>(\bigcup_{i=1}^{1000} (((1000 + i) + 5000) + 5000))</td>
<td>1000</td>
<td>41.67</td>
</tr>
</tbody>
</table>

Fig. 4. Benchmarks

\(\Sigma DD\) performs most of the time less rewriting steps thanks to sharing and caching, but the cost of single rewriting is much higher. Our implementation is far from having the maturity of Maude and thus we are confident that optimizing the cost of a single rewriting will bring much better performances. The other important point is that for practical reasons we used the so called “iter theory” in the Maude examples: \(suc(suc(suc(0))) = suc^3(0)\), this is not implemented so far in \(\Sigma DD\) and will improve both the memory footprint and processing time. The benchmarks, our implementation, as well as the specifications for Maude can be found under \url{http://smv.unige.ch/tiki-list_file_gallery.php?galleryId=59}.

As mentioned before, we primarily developed this technology to tackle the state space explosion problem that occurs when performing model checking on Algebraic Petri Net model. We use the Decision Diagrams framework to represent the states in a symbolic way. In an Algebraic Petri Net, a state is represented as a vector of places that contain multisets of terms. Since we use DD for representing set of vector of places and multisets, we have naturally extended them to support Terms. This helps to share the common sub-terms among the states and thus common rewriting steps. Using this technology, we are able to handle much bigger models than the competition (300 philosophers for AlPiNA vs. 15 for Maria [15]) as detailed in [11]. The performances (both memory footprint and processing time) are also much

\[ (5000 + 5000) + (2000 + ((500 + (500 + 500)) + 5)) \]
better. Although $\Sigma DD$ is not the only reason for such performances, it enables us to apply techniques such as algebraic clustering and unfolding as detailed in [11].

6 Related Work

Term graph rewriting theory [16,8] as well as its efficiency [17] have been intensively studied in the literature.

Several approaches leverage on sharing by using acyclic term graphs like the ATerm library (maximally shared terms) [18] to provide efficient term (graph) encoding and term (graph) rewriting. However, the novelty of our approach is that the structure is optimized not only for large terms but also for large sets of large terms. To our best knowledge, it is the only approach that focuses on handling such large sets of terms.

Since our goal is to encode sets of states containing multisets of terms, we didn’t use ATerm or Tom that is built on top of it [19], in order to maximize the sharing among the states by encoding everything in a DD-like structure.

As for ATerm, this approach does not allow side effects and thus does not suffer from the “rewrite in place” approach of Term Graph Rewriting with collapsing. Namely, some examples of non-confluent term rewriting in GTR [8] are confluent with this approach without loosing the sharing.

Unlike Tom [19] or Maude [14], we did not work on strategies so far. However only innermost and outermost strategies have been implemented, other reduction strategies can be defined using dedicated homomorphisms.

7 Conclusion and Future Work

We have presented the $\Sigma DD$ which is a powerful data structure inspired by the Decision Diagrams to encode and manipulate set of terms. The novelty of this approach resides in the strong leverage of the shared parts in set of terms to optimize both the memory footprint and the computation time. Although not as efficient as other approaches on a per rewriting basis, the huge sharing induced much less rewriting step to get to a normal form when working on large sets. We plan to extend our work in the following ways:

- More reduction strategies.
- Optimize rewriting step.
- Heuristics for efficient variable ordering.
- Extensive benchmarking with Tom [19] which is based on ATerm [18].
- Axioms between generators and Associative/Commutative rewriting.
- Add support for the so-called iter theory, namely a compressed representation of stacked operators: $\text{suc} (\text{suc}(0)) = \text{suc}^2(0)$.

We successfully used the $\Sigma DD$ in our model checker called AlPiNA, which is, freely on [http://smv.unige.ch/tiki-list_file_gallery.php?galleryId=59](http://smv.unige.ch/tiki-list_file_gallery.php?galleryId=59)
8 Acknowledgments

The authors wish to express special thanks to the members of the MoVe laboratory of University Paris VI for their valuable help and suggestions, especially to Yann Thierry-Mieg and Alban Linard. We are also grateful to the anonymous referees for their useful comments and to the Hasler Foundation for supporting this work.

References