Specification of Decision Diagram Operations

Alexandre Hamez[1], Steve Hostettler[2], Alban Linard[2], Alexis Marechal[2],
Emmanuel Paviot-Adet[3], and Matteo Risoldi[2]

[1] CNRS - LAAS, 7, avenue du Colonel Roche, 31077 Toulouse, France
`FirstName.LastName@laas.fr`

[2] Université de Genève, 7 route de Drize, 1227 Carouge, Switzerland
`FirstName.LastName@unige.ch`

[3] Université Pierre & Marie Curie, LIP6 - CNRS UMR 7606,
4 place Jussieu, 75252 Paris Cedex 05, France
`FirstName.LastName@lip6.fr`

Abstract. Decision Diagrams (DDs) are a well populated family of data structures, used for efficient representation and manipulation of huge data sets. Typically a given application requires choosing one particular category of DDs, like Binary Decision Diagrams (BDDs) or Data Decision Diagrams (DDDs), and sticking with it.

Each category provides a language to specify its operations. For instance, the operation language of BDDs provides `if-then-else`, `apply`, *etc.* We focus on two main kinds of operation languages: BDD-like and DDD-like. They overlap: some operations can be expressed in both kinds of languages, while others are only available in one kind.

We propose in this article a critical comparison of BDD-like and DDD-like languages. From the identified problems, we also propose a unified language for DD operations. It covers both BDD-like and DDD-like languages, and even some operations that cannot be expressed in either.

1 Introduction

Decision Diagrams (DDs) are now widely used in model checking as an extremely compact representations of state spaces [1]. Numerous DD categories have been developed over the past twenty years based on the same principles. Each category is adapted to a particular application domain and comes with a manipulation language, that is used to create and modify the DDs.

Typically a given application requires choosing one particular category of DDs, like Binary Decision Diagrams (BDDs) or Data Decision Diagrams (DDDs), and sticking with it. Then, the user has to learn its operations, which might be a non-trivial task. DDs are used for both efficient memory representation *and* efficient computation time. But knowing which operation leads to better efficiency sometimes requires deep knowledge and understanding of the DD category.

Two solutions try to circumvent this problem. The first one is to use high-level Domain Specific Languages (DSLs), specific to an application domain. For instance, CrocoPat[4] completely hides the DDs and provides to the user a language for manipulating relational expressions. We do *not* consider high-level languages here. The second solution is automatic optimization of low-level operations [2]. We are interested in this article only in low-level languages provided with the DDs. DSLs can then be translated to low-level programs.

Section 2 first does a brief presentation of DDs. It presents the terminology we are using in the remainder of the article, and the categories of DDs we cover.

We compare two kinds of languages, those used in BDD-like structures (BDDs, Algebraic Decision Diagrams (ADDs), Multi-valued Decision Diagrams (MDDs), for instance) in Section 3, and those used in DDD-like structures (DDDs, Set Decision Diagrams (SDDs), Σ Decision Diagrams (ΣDDs)) in Section 4. They offer very different operations to their users. Moreover, their expressiveness differ. We think they are therefore good candidates for comparison.

We then propose in Section 5 low-level operations that generalize both BDD-like and DDD-like operations. They only apply to DDs where the function's results are stored in terminal vertices, so Edge-Valued Decision Diagrams (EVDDs) are *not* covered in this article. Moreover, hierarchy as in SDDs and continuous input domains are not handled for simplicity.

2 Decision Diagram Principles

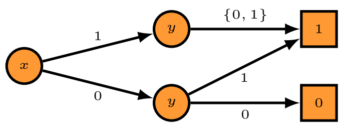
Decision Diagrams are representations of functions using Directed Acyclic Graphs (DAGs) with maximal sharing from the leaves and roots. DD categories differ on the signatures of represented functions. We note $\mathbb{B}$ the Boolean domain and $\mathbb{N}$ the natural integers. Function signatures are for instance $\mathbb{B}^n \rightarrow \mathbb{B}$ for BDDs [3], $\mathbb{B}^n \rightarrow \mathbb{N}$ for ADDs [4], and $\mathbb{N}^* \rightarrow \mathbb{B}$ (unbounded sequences of integers as inputs) for DDDs [5]. Categories also differ in the graph representation of functions. In most cases, the result is stored in terminal vertices, but Edge-Valued Binary Decision Diagrams (EVBDDs) [6] and Edge-Valued Multi-Valued Decision Diagrams (EVMDDs) [7] store them along the paths.

Represented functions are total. They return a value in their ouput domain for each value of their input domain. These domains can be infinite, either because they are discrete but not bounded as in DDDs or ADDs [4], or because they are continuous as in Interval Decision Diagrams (IDDs) [8]. As the DD is a finite graph, not all functions have a DD counterpart. This excludes functions that effectively return an infinite number of different values, as they cannot be represented by a finite number of terminals in the DD categories presented here.

A DD represents a function, which can also be seen as an association of each input element with an output one. We consider categories where every input element is a path of the graph, ending with a terminal vertex labeled by the function's corresponding output value. Each vertex on the path is a variable of the function, and edges are labeled by their possible values. An order over

[4] http://www.sosy-lab.org/~dbeyer/CrocoPat/

variables specifies the allowed succession of vertices in the graph. For instance, the Boolean function $x \vee y$ is represented by the BDD in Figure 1.



The function is described by the following set of paths:

$$\left\{ \begin{matrix} x \xrightarrow{0} y \xrightarrow{0} 0, x \xrightarrow{0} y \xrightarrow{1} 1 \\ x \xrightarrow{1} y \xrightarrow{0} 1, x \xrightarrow{1} y \xrightarrow{1} 1 \end{matrix} \right\}$$

Fig. 1. BDD for $x \vee y$ with variable order $x < y$, and its corresponding paths

We compare operations for two categories of DDs that have deep differences. First, operations on BDDs and other categories with a fixed number of variables are presented in Sec. 3. Then, Sec. 4 presents operations on DDDs where paths can use different variables, and even have different lengths. Sec.5 makes a synthesis of these two kinds, by proposing a new language for operations.

3 BDD-like Operations

BDD-like operations are used on DDs representing functions with a finite and homogeneous input domain. Their signatures are thus of the form $\mathbb{X}^n \to \mathbb{O}$, where $\mathbb{X}$ is a finite domain. This includes categories such as BDDs [3], ADDs [4], MDDs [9,10], *etc.* Their paths are therefore of the form $v_1 \xrightarrow{x_1} \dots v_n \xrightarrow{x_n} t$.

Several operations are defined for BDDs in [3]. We give their extension to finite edges [10] and terminal [4] domains. In practice, they are used in BDD libraries, such as CUDD[5].

The following operations form the BDD-like manipulation language. It is composed of functions, which take DDs (noted d_i), variables (v_i), input values (x_i), output values (t_i) or operators ($\odot$) as parameters. We give – informally – the result of each operation as its set of returned paths.

`constant(`t`)` creates a DD where all paths lead to the terminal value t. It represents the constant function that returns t whatever its inputs are.

`constant(`t`)=` $\left\{ v_1 \xrightarrow{x_1} \dots v_n \xrightarrow{x_n} t \right\}$[6]

`make(`v`, `x`)` creates a DD where only the variable v is relevant to the function's result. The paths where v has value x lead to terminal value 1, and others to terminal 0, even for DD categories with unbounded terminals such as ADDs. We explain this restriction in Problem 1.

`make(`v`, `x`)=` $\left\{ v_1 \xrightarrow{x_1} \dots v \xrightarrow{\mathbb{X} \setminus \{x\}} \dots v_n \xrightarrow{x_n} 0 \right\} \cup \left\{ v_1 \xrightarrow{x_1} \dots v \xrightarrow{x} \dots v_n \xrightarrow{x_n} 1 \right\}$

`apply(`$\odot$`, `d_l`, `d_r`)` computes $d_l \odot d_r$. It takes two DDs d_l and d_r representing functions of signature $\mathbb{X}^n \to \mathbb{O}$, and a binary operator $\odot : \mathbb{O} \times \mathbb{O} \to \mathbb{O}$ on their output domain. It returns the DD of the function computed by:

`apply(`$\odot$`, `d_l`, `d_r`)=` $\left\{ v_1 \xrightarrow{x_1} \dots v_n \xrightarrow{x_n} t_1 \odot t_2 \;\middle|\; \begin{matrix} v_1 \xrightarrow{x_1} \dots v_n \xrightarrow{x_n} t_1 \in d_l \\ v_1 \xrightarrow{x_1} \dots v_n \xrightarrow{x_n} t_2 \in d_r \end{matrix} \wedge \right\}$

[5] http://vlsi.colorado.edu/~fabio/CUDD/

[6] We omit universal quantification on x_i for readability, as in other operations.

`restrict(`v`, `x`, `d`)` restricts d to the value x of the variable v, and then ignores this variable's values. The returned function's result is similar to the original one, when v has value x, so this operation computes $f|_{v=x}$. The resulting DD still has all variables, including v, which has no effect on its result.
`restrict(`v`, `x`, `d`)`$= \left\{ v_1 \xrightarrow{x_1} \dots v \xrightarrow{\mathbb{X}} \dots v_n \xrightarrow{x_n} t \middle| v_1 \xrightarrow{x_1} \dots v \xrightarrow{x} \dots v_n \xrightarrow{x_n} t \in d \right\}$

`compose(`v`, `d_l`, `d_r`)` substitutes the variable v in d_l with the DD d_r. Both DDs d_l and d_r must represent functions of type $\mathbb{X}^n \rightarrow \mathbb{X}$. This operation can be expressed using `apply` and `restrict`, and does not appear in articles on BDDs other than [3].

`satisfy-all(`t`, `d`)` returns the set of paths with terminal value t. The result of this operation is *not* a DD but an iterable enumeration of paths.
`satisfy-all(`t`, `d`)`$= \left\{ v_1 \xrightarrow{x_1} \dots v_n \xrightarrow{x_n} t \in d \right\}$

`satisfy-one(`t`, `d`)` chooses one element from those returned by `satisfy-all`. Choice may be deterministic or not, depending on the implementation.

`satisfy-count(`t`, `d`)` Instead of returning paths, this operation counts them. Again, the result is *not* a DD, it is an integer.
`satisfy-count(`t`, `d`)`$= \left| \left\{ v_1 \xrightarrow{x_1} \dots v_n \xrightarrow{x_n} t \in d \right\} \right|$

The language for BDD-like operations is widely used, in many libraries and applications. The example given below[7] builds a BDD for $\overline{x} \wedge \overline{y} \wedge \overline{z}$. It then restricts the function for variable $x = 0$ and counts the paths leading to terminal 1. Results of operations are given as comments, after `//`. Notice that all operations, except `satisfy-count`, return a DD containing *all* variables x, y, z. Their declaration is required before executing the operation. As the operation is composed of nested function calls, it should be read from bottom to top. The call to `constant` is common in BDD programming, but not useful here.

```
satisfy-count // $\Longrightarrow 2$
( 1
, restrict // $\Longrightarrow x \xrightarrow{0,1} y \xrightarrow{0} z \xrightarrow{0} 1 \cup \dots \rightarrow 0$
  ( $x$, 0
  , apply // $\Longrightarrow x \xrightarrow{0} y \xrightarrow{0} z \xrightarrow{0} 1 \cup \dots \rightarrow 0$
    ( $\wedge$
    , make($z$, 0) // $\Longrightarrow x \xrightarrow{0,1} y \xrightarrow{0,1} z \xrightarrow{0} 1 \cup x \xrightarrow{0,1} y \xrightarrow{0,1} z \xrightarrow{1} 0$
    , apply // $\Longrightarrow x \xrightarrow{0} y \xrightarrow{0} z \xrightarrow{0,1} 1 \cup \dots \rightarrow 0$
      ( $\wedge$
      , make($y$, 0) // $\Longrightarrow x \xrightarrow{0,1} y \xrightarrow{0} z \xrightarrow{0,1} 1 \cup x \xrightarrow{0,1} y \xrightarrow{1} z \xrightarrow{0,1} 0$
      , apply // $\Longrightarrow x \xrightarrow{0} y \xrightarrow{0,1} z \xrightarrow{0,1} 1 \cup x \xrightarrow{1} y \xrightarrow{0,1} z \xrightarrow{0,1} 0$
        ( $\wedge$
        , make($x$, 0) // $\Longrightarrow x \xrightarrow{0} y \xrightarrow{0,1} z \xrightarrow{0,1} 1 \cup x \xrightarrow{1} y \xrightarrow{0,1} z \xrightarrow{0,1} 0$
        , constant(1) // $\Longrightarrow x \xrightarrow{0,1} y \xrightarrow{0,1} z \xrightarrow{0,1} 1$
) ) ) ) )
```

[7] Inspired by the documentation of CUDD.

This language for BDD-like operations has several drawbacks. We describe them in the remainder of this section. Note that these drawbacks do not prevent wide use of this language.

Problem 1 (`make` restricts terminal values). Creating a DD with this operation requires a default value for other paths. For instance, `make(`x`, 0)` creates a DD returning 1 whenever $x = 0$, but implicitly also 0 for the other cases.

Boolean functions have a trivial implicit result, as only two terminal values are available. But it is not the case for ADDs, which represent functions $\mathbb{B}^n \to \mathbb{N}$. Their implicit result is chosen as the identity element 0 of addition. With addition, all terminal values can be obtained, as in `apply(`$+$`, make(`x`, 0), make(`x`, 0))` which returns $x \xrightarrow{0} y \xrightarrow{0,1} z \xrightarrow{0,1} 2 \cup x \xrightarrow{1} y \xrightarrow{0,1} z \xrightarrow{0,1} 0$.

So, the set of terminal values should be a monoid for BDD-like operations. It requires defining a $+$ operation to generate all possible terminal values. Note that `constant` can then be also restricted to the terminals' generator only.

Problem 2 (The language is not minimalist). The `compose` operation is redundant. It can be expressed using two other operations, `apply` and `restrict`. Usually, this is not a problem, as redundant operations can be safely removed from the language to get a minimal one. But the operation is specifically defined because its algorithm is more efficient. So, this BDD-like language is not minimalist, but removing the redundant operation has an efficiency cost.

The same task can be defined in several ways, that are not equivalent regarding efficiency. Writing a complex operation therefore requires knowledge from the user on how its evaluation internally works.

Problem 3 (The language requires embedding). A usual operation in model checking is to compute the fixed point of next states computation. But the BDD-like language has no operations for iteration or recursion. So, it requires to be *embedded* into a general-purpose language, that can compute the fixed point.

The BDD-like languages are thus truly DSLs. They describe an operation in a concise and readable manner, but are not as expressive as a general-purpose programming language. They can express only very simple computations.

Problem 4 (Operations cannot be optimized). In [11] it has been shown that rewriting the operations can lead to huge performance improvements. This has been partially automated in DDD-like languages [2]. Because of embedding (see Problem 3), a library with BDD-like operations cannot see the whole operation to perform. It only processes it in small operations. Only few optimizations are available for these, whereas most improvements require access to the full syntax tree of the operation.

This problem is very similar to Problem 2, which occurs because the `compose` operation cannot be expressed efficiently enough using other operations. The language is not low-level enough to enable optimizations that could bring the same efficiency. Its operations take whole DDs as parameters and return a whole DD. So, the language lacks more intrusive operations, that apply to parts of the DDs. DDD-like operations presented in Section 4 define them, and have shown great optimizations thanks to lower-level operations.

4 DDD-like Operations

The main difference between BDD-like and DDD-like structures is the input domain of represented functions. In [5], paths are defined as "sequences of assignments", because in the same DD, paths can be of various lengths, and the same variable can appear several times along a path. We cannot describe the domain of functions represented by DDD-like structures with a cartesian product notation. We propose in [12] a specification of their domain.

These categories include DDDs [5] and SDDs [13] which represent sets and have Boolean terminals, and MultiSet Decision Diagrams (MSDDs) [14] which represent multisets with natural integer terminals. We propose in [12] a unification of BDD-like and DDD-like structures, where they represent functions on complex data types rather than functions with multiple arities. We do not give importance to these subtleties in this article, so readers only need to be aware of paths with different lengths and variable repetition.

DDD-like operations are partitioned in two languages: a language of binary set operations (Sec. 4.1) and a language of unary set transformation (Sec. 4.2). They are not totally disjoint, as bridges exist between them.

4.1 Binary Set Operations

The language of set expressions is composed of the usual binary set operations $\cup, \cap, \setminus$ – or their multiset counterparts – applied to two DDs. These three operations are distinct, whereas they are all covered by `apply` in BDD-like operations. A general definition is difficult, because domains of variables can be unbounded. To preserve the finite graph representation of the function, an infinite number of paths must lead to terminal 0, whereas a finite number lead to other terminals. So, the operation applied to terminals should always return 0 when both its operands are 0 to ensure termination. A general `apply` could exist, given operator on terminals respects this constraint.

These operations are a small subset of BDD-like operations. For instance, they are not sufficient to compute a state space. So, they are completed with a language for set transformations.

4.2 Unary Set Transformations

The unary set transformations, initially proposed in [5], have nothing in common with BDD-like operations. They are close to the `map` functions found in most functional languages: for a DD category representing sets, an operation is applied to each path. But instead of returning a transformed path, application on each path returns a set of paths, each set represented by a DD.

These transformations are homomorphisms. They must therefore enforce a constraint: all must be linear for a union $\cup$ operator defined in the DD category: $h(d_1 \cup d_2) = h(d_1) \cup h(d_2)$. The linearity property means that every input DD can be split in several DDs, then the operation applied to each, and the result obtained by union of their results. For simplicity, operations given below do not mention particular cases required by linearity. These cases are common to all operations, and are discussed in Problems 8 and 9. Note that these operations are applied on quasi-reduced DDs, *i.e.* without level skipping.

Unary transformations are parametric: they take parameters that define their behavior in addition to the DD to transform. To describe the language of such operations, we show each one with both kinds of parameters. The first one, between brackets, contains all parameters except the DD to transform. The second one, between parentheses, is this DD. We thus clearly distinguish between parameters that define the operation, and the parameter to which it is applied.

$\texttt{terminal[}t\texttt{]}$ is the only operation that allows to create a DD outside unary set transformations. It is therefore crucial, as the basis for each operation. This nullary operation creates a DD terminal valued by t. It is close to the `constant` BDD-like operation. But they differ because the DD returned by `terminal` represents the nullary function that returns t, whereas the DD returned by `constant` represents a n-ary function.
$\texttt{terminal[}t\texttt{]}() = \{\mapsto t\}$

$\texttt{constant[}d_c\texttt{]}$ returns the constant DD d_c, whatever its input DD is. This operation is parameterized by the DD to return, and is applied to a DD that it ignores (almost, see Problem 8). The name of this operation can be misleading. The returned DD represents a function that can be constant or not: all its paths do not necessarily lead to the same terminal. But the `constant` operation returns this DD independently of its input. It thus differs from `constant` of BDD-like operations, which returns a constant function.
$\texttt{constant[}d_c\texttt{]}(d) = d_c$

`identity[]` returns its input with no modification. This operation can be expressed using other operations, but is introduced in DDDs for efficiency.
$\texttt{identity[]}(d) = d$;

$\texttt{make[}v \xrightarrow{x}\texttt{]}$ adds a prefix $v \xrightarrow{x}$ to a DD. It differs from the BDD-like `make` operations, because it adds a variable to the represented function given as parameter, whereas the BDD-like creates an n-ary function from scratch.

$$\texttt{make[}v \xrightarrow{x}\texttt{]}(d) = \left\{ \begin{array}{c} v \xrightarrow{x} v_1 \xrightarrow{x_1} \ldots v_n \xrightarrow{x_n} t \\ v \xrightarrow{\{y \neq x\}} v_1 \xrightarrow{x_1} \ldots v_n \xrightarrow{x_n} 0 \end{array} \middle| \; v_1 \xrightarrow{x_1} \ldots v_n \xrightarrow{x_n} t \in d \right\}$$

$\texttt{match[}v \xrightarrow{x}\texttt{, } h\texttt{]}$ is the inverse operation of `make`. It selects paths where $v \xrightarrow{x}$ is a prefix of the DD, and then applies another operation h to the subDD that prefix leads to. Other paths return the identity DD.
$$\texttt{match[}v \xrightarrow{x}\texttt{, } h\texttt{]}(d) = h\left(\left\{v_1 \xrightarrow{x_1} \ldots v_n \xrightarrow{x_n} t \;\middle|\; v \xrightarrow{x} v_1 \xrightarrow{x_1} \ldots v_n \xrightarrow{x_n} t \in d\right\}\right)$$
This operation is overloaded, as it also exists to match a terminal.
$\texttt{match[}t\texttt{, } h\texttt{]}(d) = h\left(\{t \mid t \in d\}\right)$

$\texttt{composition[}h_1\texttt{, } h_2\texttt{]}$ computes the composition $h_1 \circ h_2$, and applies it to its input DD. It has no relation with `compose` of BDD-like operations, which substitutes a variable with the result of a DD.
$\texttt{composition[}h_1\texttt{, } h_2\texttt{]}(d) = (h_1 \circ h_2)(d) = h_1(h_2(d))$

`union`[h_1, h_2] (respectively `intersection`) computes the union (resp. intersection) of results of h_1 and h_2 applied to the input DD. This operation wraps some of the set operations presented in Sec. 4.1. Set difference $\setminus$ is not wrapped because it is not a linear operation. Note that these two operations can be easily extended to n-ary versions as they are associative and commutative.
`union`[h_1, h_2](d) $= h_1(d) \cup h_2(d)$ `intersection`[h_1, h_2](d) $= h_1(d) \cap h_2(d)$

`fixpoint`[h] has been introduced for model checking applications. Problem 3 shows that BDD-like languages lack operations for iteration. `fixpoint` provides such an operation. It computes the fixed point of h, applied to the input DD.
`fixpoint`[h](d) $= h^{\infty}(d) = h(\ldots h(d) \ldots)$

The DDD-like unary set transformations are much more expressive than the BDD-like operations. Figure 2 shows a Petri net and an operation that computes its state space, which will be improved later. The operation uses the `fixpoint` operator where BDD-like operations require to be embedded in a general purpose language. To encode an least fixed point, `identity` is used to keep previously computed states.

We show how the operation works by applying the operation for transition t to the DDD path $q \xrightarrow{1} p \xrightarrow{2} r \xrightarrow{0} 1$ representing the initial state of the Petri net.

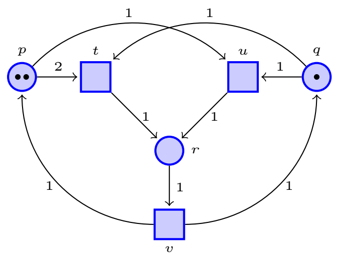
$$
\begin{array}{rr}
 & \texttt{match}[q \xrightarrow{1}, \ldots]\ (q \xrightarrow{1} p \xrightarrow{2} r \xrightarrow{0} 1) \\
\Longrightarrow & \texttt{make}[q \xrightarrow{0}, \ldots]\ (p \xrightarrow{2} r \xrightarrow{0} 1) \\
\Longrightarrow & q \xrightarrow{0} \texttt{match}[p \xrightarrow{2}, \ldots]\ (p \xrightarrow{2} r \xrightarrow{0} 1) \\
\Longrightarrow & q \xrightarrow{0} \texttt{make}[p \xrightarrow{0}, \ldots]\ (r \xrightarrow{0} 1) \\
\Longrightarrow & q \xrightarrow{0} p \xrightarrow{0} \texttt{match}[r \xrightarrow{0}, \ldots]\ (r \xrightarrow{0} 1) \\
\Longrightarrow & q \xrightarrow{0} p \xrightarrow{0} \texttt{make}[r \xrightarrow{1}, \ldots]\ (1) \\
\Longrightarrow & q \xrightarrow{0} p \xrightarrow{0} r \xrightarrow{1} \texttt{match}[1, \ldots]\ (1) \\
\Longrightarrow & q \xrightarrow{0} p \xrightarrow{0} r \xrightarrow{1} \texttt{identity}[]\ (1) \\
\Longrightarrow & q \xrightarrow{0} p \xrightarrow{0} r \xrightarrow{1} 1
\end{array}
$$

Problem 5 (Operations usually must be lazily defined). The operation given in Figure 2 has a bug, because the `match` operation for transition u requires exactly one token in p. So, it cannot be applied from the initial state. We have to enumerate all possible cases, by adding :

, `match`[$q \xrightarrow{1}$, `make`[$q \xrightarrow{0}$, `match`[$p \xrightarrow{1}$, `make`[$p \xrightarrow{0}$,
`match`[$r \xrightarrow{0}$, `make`[$r \xrightarrow{1}$, `match`[1, `identity`[]...] `// u`

We cannot express "add n tokens" or "remove n tokens", only "set to n tokens". In practice, an operation usually has an *a priori* unbounded number of `match` calls. They will be discovered during computation. Description of the operation would be infinite, so it is in practice rather defined lazily. Then `match` uses a function returning the operation associated with each encountered value.

Problem 6 (Lazy operations cannot be optimized). The operations that compute the state space of Petri net in Figure 2 can be optimized. The operation



```
fixpoint[
  union[
    identity
  , match[$q \xrightarrow{1}$, make[$q \xrightarrow{0}$, match[$p \xrightarrow{2}$, make[$p \xrightarrow{0}$,
    match[$r \xrightarrow{0}$, make[$r \xrightarrow{1}$, match[1, identity[]...] // t
  , match[$q \xrightarrow{1}$, make[$q \xrightarrow{0}$, match[$p \xrightarrow{1}$, make[$p \xrightarrow{0}$,
    match[$r \xrightarrow{0}$, make[$r \xrightarrow{1}$, match[1, identity[]...] // u
  , match[$q \xrightarrow{0}$, make[$q \xrightarrow{1}$, match[$p \xrightarrow{0}$, make[$p \xrightarrow{1}$,
    match[$r \xrightarrow{1}$, make[$r \xrightarrow{0}$, match[1, identity[]...] // v
] ]
```

Fig. 2. A Petri net and the operation encoding its firing rule. The variable order used by operations is $q < p < r$. Transitions' names are given as comments after their encoding. We use `]...]` instead of numerous closing brackets, so indentation is meaningful.

is rewritten to an equivalent but more efficient one, given below, by merging identical `match` calls. It is more efficient because some parts of the operation are merged, and thus require fewer application of the operations. The currently most advanced optimization technique, called "automatic saturation" has been introduced in [2]. It follows the same principle of rewriting an operation into a more efficient one.

```
fixpoint[
  union[
    identity
  , match[$q \xrightarrow{1}$, make[$q \xrightarrow{0}$, union[
      match[$p \xrightarrow{1}$, make[$p \xrightarrow{0}$, match[$r \xrightarrow{0}$, make[$r \xrightarrow{1}$,
      match[1, identity[]...]
    , match[$p \xrightarrow{2}$, union[
        make[$p \xrightarrow{0}$, match[$r \xrightarrow{0}$, make[$r \xrightarrow{1}$, match[1, identity[]...]
        make[$p \xrightarrow{1}$, match[$r \xrightarrow{0}$, make[$r \xrightarrow{1}$, match[1, identity[]...]
    ]...]
  , match[$q \xrightarrow{0}$, make[$q \xrightarrow{1}$, match[$p \xrightarrow{0}$, make[$p \xrightarrow{1}$,
    match[$r \xrightarrow{1}$, make[$r \xrightarrow{0}$, identity[]...]
] ]
```

Non-optimized and optimized ones are expressed in the same language. So, the language is suitable for optimization by program transformation. It solves the Problem 4 of BDD-like languages.

Lazy operations seen in Problem 5 are useful to deal with *a priori* unbounded structures. But they have a drawback: they cannot be optimized, because their suboperations are not known until execution. To circumvent the problem, new operations have been added in DDD-like languages. They are redundant with existing ones, but provide the missing information. However, as more optimizations are added, more specific operations like these must also be added. This approach creates several redundant ways to define the same operations. Users have to know which one is better to enable most optimizations.

Problem 7 (Weak typing). Each operation is designed to work on DDs of a given type, *i.e.* a variable order and a category that defines the variables' domains and output domain. For instance, the following operation can only be applied to DDDs of variable order $x < y$. It detects the path $x \xrightarrow{1} y \xrightarrow{1} 1$.

```
match[x -1->, match[y -1->, match[1, constant[terminal[1]]]]]
```

Operations' operands which are DDs are not explicitly typed. Only the user knows their type. So, an operation can discover at runtime that it is not applied to an intended operand. The operation then fails, but how this failure is reported is unclear in the DDD-like languages: return of a special terminal $\top$, assertion to stop the program, exception that can be caught. . . or even return of terminal 0 which is in fact silent.

Instead of giving a DD of the wrong type to an operation, typing errors can also appear because of DDs returned by inner operations. This leads to errors (known in DDDs as the "top" $\top$ terminal) that are hard to debug[8]. Such a typing error occurs when operands of `union` or `intersection` do not return DDs of the same type. For instance, the following operation is erroneous because the first part of `union` returns a DDD with variable order $x < y$ whereas the second has variable order $y < x$.

```
union[ make[x -1->, make[y -0->, constant[terminal[1]]]
     , make[y -1->, make[x -0->, constant[terminal[1]]]]]
```

It seems easy here to check the typing problems. But lazy operations, required for Problem 5, prevent checking in most practical cases.

Problem 8 (An identity DD is only transformed to an identity DD). The linearity constraint of homomorphisms requires that every DD can be split in several other DDs, operations applied to them, and their results merged.

Each DD can be decomposed to itself and the identity DD (noted d_0), where all paths lead to terminal 0. From the definition of linearity, a DD can be decomposed in itself and the identity, as in $h(d) = h(d \cup d_0) = h(d) \cup h(d_0)$. So, every operation must return the identity DD when applied to the identity d_0.

[8] Users that have used at least once a DDD library know the pain to look at thousands of lines of DD paths to search where something went wrong.

For instance, all operations applied to the DDD representing the empty set return the empty set. Even `constant[`d_c`]`(d_0) cannot return its DD parameter d_c when applied to d_0. In this particular case, it must return the identity d_0. This constraint does not exist in BDD-like operations. They can return any DD when applied to the identity one.

This requirement prevents the user of DDD-like operations to start its application with the identity DD, and then filling it through operations. In model checking applications using DDDs, we have to start therefore with an initially non-empty set of states, which is usually the initial state.

Problem 9 (Operations on terminals must also be linear). This is another consequence of homomorphisms, as in Problem 8. Consider a DD representing multisets. Each path represents an element of the multiset, its terminal represents its cardinality. The multiset union sums the terminals of identical paths. Because each DD can be decomposed, the result computed for a path leading to n must be the same as the sum of results for the same path ending with n_1 and n_2, where $n = n_1 + n_2$.

For instance, Fig. 3 shows an operation that tries to count paths leading to terminal 1. Its result is computed by adding all subresults, given as destinations of red bold arrows. The final result is shown for a DD (left part) and for one arbitrary decomposition of the DD (right part). They differ, because the operation on terminals is not linear.

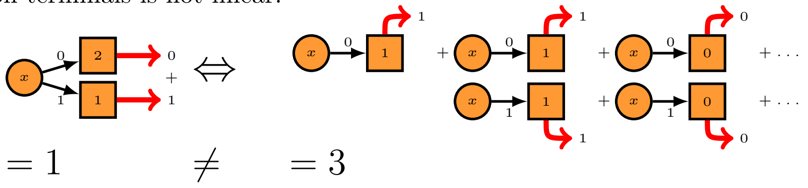


Fig. 3. Counting paths leading to terminal 1 is not a linear operation

Users specify operations only for paths leading to 1 in DDDs. This category has Boolean terminals, and homomorphisms always return the identity DD d_0 when paths end with 0. For DDs representing multisets (MSDDs in [14]), letting the user specify the operation for any non-zero terminal value is dangerous, as it might break linearity. To ensure this property, user should still define the operation only for terminal 1.

Problem 10 (Goal of restriction to linear operations is unclear). Even if we allow operations to return not only DDs, but also other data types such as integers, the linearity constraint is too restrictive for many operations. For instance, even a simple operation like counting the number of paths leading to a particular terminal value is *not* a linear operation (see Fig. 3 of Problem 9). DDD-like languages have an operation to count paths, but it cannot be used within linear operations. So, this special operation is doomed to appear only at the end of a computation. The reason why DDDs define their operations as homomorphisms is not explicitly given in [5]. They are well adapted for set transformation, but they restrict drastically the expressiveness of their language. We believe that the main justification for such constraints is efficiency.

Linearity, $h(d_1 \cup d_2) = h(d_1) \cup h(d_2)$, ensures not only that a DD can be decomposed to its paths to compute the operation's result, but also that the subresults can always be combined *before* the next computation. Consider the operation $(f \circ g)(d_1 \cup d_2)$. It can be rewritten as the union of two compositions $(f \circ g)(d_1) \cup (f \circ g)(d_2)$, or as $f(g(d_1) \cup f(g(d_2))$. In both cases, the result if the same, whether f is applied to two subresults $g(d_1)$ and $g(d_2)$, or to their union $g(d_1) \cup g(d_2)$. The second case is likely to do less computations[9] than the first one, thus saving time and memory.

This optimization is possible because of the linearity constraint. We propose in Sec. 5 to move this constraint where it should be: as an optimization enabled only when operations can be proven linear.

Problem 11 (Unary operations are not enough). Some operations, such as set union $\cup$, intersection $\cap$ or difference $\setminus$ for DDDs are not unary. As DDD-like set transformations are unary only, there is no way for a user to describe them inside the homomorphisms framework.

Restriction to unary functions is useful to define operations as combinators. For instance, `composition[match[`$x \xrightarrow{1}$`, identity], make[`$y \xrightarrow{0}$`, identity]]` is a unary operation, because each suboperation is also unary and returns exactly one result. The input DD argument of composed functions is implicit.

Composition of n-ary operations does not allow such shortcuts, because several links can exist between the inputs and outputs of composed operations. But n-ary operations are the general case, that is missing in DDD-like operations.

5 Generalized Operations

We propose in this section a language for generalized operations, that solves the problems identified in Sections 3 and 4. This language covers both BDD-like and DDD-like languages. Throughout the section, a simple program given in Fig. 4 is used as example. It counts the similar paths in two DDs. Note that the proposed operations are applied to quasi-reduced DDs, as in DDD-like unary operations. Missing parts must be rebuilt on-the-fly when the operations are applied.

Program A program is a set of *named* operations, like *check*, $check_y$, $check_{x<y}$ and *count* in Fig. 4. Naming enables recursion. There is therefore no need for a particular `fixpoint` operation, nor embedding loops in a general-purpose language (Problem 3). One of these named operations is called by the user.

Typed inputs and outputs Each named operation has *typed* inputs and outputs (Problem 7). For instance, *count*`(`l`: in `t`, `r`: in `t`, `o`: out `*int*`)` has two inputs l and r of the same type t, and one output o of type *int*. To handle n-ary operations (Problem 11), several – or no – inputs and outputs are possible. All of them are DDs, for homogeneity, even values (*int*) are considered as terminals. Operand types are defined by a DD category (BDD, DDD, *etc.*) and a variable order. We

[9] But in DDs, who knows?

```
check (l: in v, r: in v, o: out int) =
  match | l: |t|, r: |t| ⇒ o ←make |1|
        | l: |t|, r: |u| ⇒ o ←make |0|
check_y (l: in u, r: in u, o: out int) =
  match | $l : y \xrightarrow{v} d_l, r : y \xrightarrow{v} d_r \Rightarrow check(l \leftarrow d_l, r \leftarrow d_r, o \rightarrow o)$
        | $l : y \xrightarrow{u} d_l, r : y \xrightarrow{v} d_r \Rightarrow o \leftarrow$make |0|
check_{x<y} (l: in t, r: in t, o: out int) =
  match | $l : x \xrightarrow{v} d_l, r : x \xrightarrow{v} d_r \Rightarrow check_y(l \leftarrow d_l, r \leftarrow d_r, o \rightarrow o)$
        | $l : x \xrightarrow{u} d_l, r : x \xrightarrow{v} d_r \Rightarrow o \leftarrow$make |0|
count (l: in t, r: in t, o: out int) =
  merge o from $check_{x<y}(l \leftarrow l, r \leftarrow r, o)$ with +
```

Fig. 4. Program that counts the similar paths in two DDs with variables $x < y$

propose in [12] a way to define types with an automaton describing the domains of variables and their order.

We use four types of DDs in our example. Three types correspond to DDs with Boolean terminals and unbounded variables: no variable (type v), variable y (type u) and variables $x < y$ (type t). The fourth type corresponds to DDs with natural terminals and no variables (type int), used to count the results.

Predefined operation templates Operations can be of three kinds, that we call "operation templates". They are inspired from DDD-like operations, but are minimalist as each one has a very specific role. They thus solve Problem 2.

`match` | $prefix^n \Rightarrow call$ | ... takes as parameter a finite mapping from prefix patterns to operation calls. As we deal with n-ary operations, a pattern is composed of n prefixes, one for each input operand. For each set of DD paths, one taken from each operand, this operation finds the first matching pattern, and applies the associated operation. A prefix can be:

- a terminal constant or placeholder[10] for instance |1| or |t|,
- a DD placeholder (which also covers a terminal), for instance d,
- a prefix with constants or placeholders as variable or/and edge values, and another prefix for the successor, for instance $1 \xrightarrow{x} d$, $x \xrightarrow{0} d$, or $x \xrightarrow{y}$|t|.

As in functional languages with pattern-matching, all the patterns of a `match` must cover all possible prefixes of the input DDs. They also must bind all the outputs of the operation. The following binary operation, which uses the value of x as discriminant, is only valid when x has a Boolean domain.

`match` | $d_1 : x \xrightarrow{1} d, d_2 : x \xrightarrow{y} d' \Rightarrow \ldots$ | $d_1 : x \xrightarrow{0} d, d_2 : x \xrightarrow{y} d' \Rightarrow \ldots$

d_i are the names of input parameters that are matched against a prefix, and "..." is here the next operation to perform (not given). Scope of placeholders is in their pattern, so the two ys are distinct in the previous example, but the two xs in a pattern $x \xrightarrow{x}$ have the same value. The following unary operation is valid even for unbounded domains, as placeholder y can take any value.

[10] We name terminal placeholder a language variable that can store a terminal value. The term "variable" is already used in DDs, so we do not use it to avoid ambiguities.

`match | ` $d_1 : x \xrightarrow{x} d \Rightarrow \dots$ ` | ` $d_1 : x \xrightarrow{y} d \Rightarrow \dots$

Note that patterns overlap in the previous examples. We chose the usual policy of functional languages: only the first matching pattern is applied. This is useful to define default cases.

Operation calls bind inputs and outputs of caller and callee. We can wrap a placeholder with an expression, to compute a value from its content. However, we do not describe expressions here, by lack of space. So, placeholders are simply bound to inputs and outputs of operations.

`make |` *pattern* `|` ... creates a DD from patterns using constants and placeholders. Patterns are expressed in the same way as in `match`. They must also cover all possible prefixes of created DD. In Fig.4, we only create terminals, but this operation can also create DD nodes. For instance, the code below creates the DD $x \xrightarrow{0} 0 \cup x \xrightarrow{1} 1$, when d is a placeholder holding the terminal 0, x is a variable and y is a free placeholder of Boolean domain. Placeholders used in the patterns can be bound to a value or be free. The latter case is useful to fill a whole unbounded domain.

`make | ` $x \xrightarrow{0} d$ ` | ` $x \xrightarrow{y}$ `|1|`

As patterns may overlap, we chose a policy consistent with `match`: when a prefix can be created using several patterns, the first one prevails. Note that this operation template does not require a default value, and thus solves Problem 1.

`merge` *placeholder* `from` *call* `with` *operator* is a new operation. It is required because there is no more a global $\cup$ operator as in DDD-like languages. Each `match` operation does not return a single DD, but a multiset of DDs, because it generates one or several DDs for each matching prefix. `merge` maps operations to how their results should be merged. The operator used to merge results ($+$ in Fig 4) is specified only for terminals, following the BDD-like `apply` mechanism. It is applied on terminals of paths that differ only by their terminal values.

This operator must be associative and commutative, because there is no order on generated DDs. Moreover, when dealing with unbounded domains, it must have an identity element. Homomorphisms are a special case, where the merge operator respects linearity.

6 Conclusion

This article does a critical comparison of BDD-like and DDD-like manipulation languages. It also proposes a new language for manipulation of DDs, that takes inspiration from both BDD-like languages and DDD-like languages, and covers them. This language also enables more operations. We give the example of counting paths that are similar in two DDs. This operation cannot be defined in the BDD-like and DDD-like languages, but is available in our proposal. The proposed language solves the problems we identify, such as minimalism of the language or expressiveness. It provides a small number of operations that have been carefully designed to allow future optimizations and are general enough to handle n-ary typed operations.

We plan to extend the proposed language to hierarchical DDs, as they are useful DD categories. Moreover, we have to redefine the optimizations that already exist for SDDs. Adding some kind of genericity to the language is also required, because the proposed specification of types can prevent code reuse.

References

1. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic Model Checking: 10^{20} States and Beyond. In: LICS '90: 5th Annual IEEE Symposium on Logic in Computer Science. (1990) 428–439
2. Hamez, A., Thierry-Mieg, Y., Kordon, F.: Hierarchical Set Decision Diagrams and Automatic Saturation. In: Petri Nets '08: 29th International Conference on Applications and Theory of Petri Nets. (2008) 211–230
3. Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers **35**(8) (1986) 677–691
4. Bahar, R.I., Frohm, E.A., Gaona, C.M., Hachtel, G.D., Macii, E., Pardo, A., Somenzi, F.: Algebraic Decision Diagrams and Their Applications. Formal Methods in System Design **10**(2/3) (1993) 188–191
5. Couvreur, J.M., Encrenaz, E., Paviot-Adet, E., Poitrenaud, D., Wacrenier, P.A.: Data Decision Diagrams for Petri Net analysis. In: ICATPN '02: 23rd International Conference on Applications and Theory of Petri Nets. (2002) 101–120
6. Lai, Y., Sastry, S.: Edge-Valued Binary Decision Diagrams for Multi-Level Hierarchical Verification. In: DAC '92: 29th ACM/IEEE conference on Design automation. (1992) 608–613
7. Ciardo, G., Siminiceanu, R.: Using Edge-Valued Decision Diagrams for Symbolic Generation of Shortest Paths. In: FMCAD '02 : 4th International Conference on Formal Methods in Computer-Aided Design. (2002) 256–273
8. Strehl, K., Thiele, L.: Symbolic Model Checking of Process Networks Using Interval Diagram Techniques. In: ICCAD '98: 1998 IEEE/ACM international conference on Computer-aided design. (1998) 686–692
9. Srinivasan, A., Kam, T., Malik, S., Brayton, R.K.: Algorithms for Discrete Function Manipulation. In: ICCAD '90: International Conference on CAD. (1990) 92–95
10. Corella, F., Zhou, Z., Song, X., Langevin, M., Eduard, C.: Multiway Decision Graphs for Automated Hardware Verification. Formal Methods in System Design **10**(1) (1997) 7–46
11. Ciardo, G., Lüttgen, G., Siminiceanu, R.: Saturation: An Efficient Iteration Strategy for Symbolic State-Space Generation. In: TACAS '01: Tools and Algorithms for the Construction and Analysis of Systems. (2001) 328–342
12. Linard, A., Paviot-Adet, E., Kordon, F., Buchs, D., Charron, S.: polyDD: Towards a Framework Generalizing Decision Diagrams. In: ACSD'10: 10th International Conference on Application of Concurrency to System Design. (2010)
13. Couvreur, J.M., Thierry-Mieg, Y.: Hierarchical Decision Diagrams to Exploit Model Structure. In: FORTE '05 : Formal Techniques for Networked and Distributed Systems. (2005) 443–457
14. Lucio, L., Hostettler, S.: Multi-set decision diagrams. Technical Report 205, Centre Universitaire D'Informatique, Université de Genève (2009) Available as http://smv.unige.ch/technical-reports/TR205-MSDD.pdf.