polyDD: Towards a Framework Generalizing Decision Diagrams

Alban Linard[*†‡] Emmanuel Paviot-Adet[†] Fabrice Kordon[†] Didier Buchs[*] Samuel Charron[‡]
[*]Université de Genève, 7 route de Drize, 1227 Carouge, Switzerland
Email: Alban.Linard@unige.ch, Didier.Buchs@unige.ch
[†]Université Pierre & Marie Curie, LIP6 - CNRS UMR 7606, 4 place Jussieu, 75252 Paris Cedex 05, France
Email: Emmanuel.Paviot-Adet@lip6.fr, Fabrice.Kordon@lip6.fr
[‡]EPITA Research and Development Laboratory (LRDE), 14-16, rue Voltaire, 94276 Le Kremlin-Bicêtre Cedex, France
Email: Samuel.Charron@lrde.epita.fr

***Abstract*—Decision Diagrams are now widely used in model checking as extremely compact representations of state spaces. Many Decision Diagram categories have been developed over the past twenty years based on the same principle. Each one targets a specific domain with its own characteristics. Moreover, each one provides its own definition. It prevents sharing concepts and techniques between these structures.**

This paper aims to propose a basis for a common Framework for Decision Diagrams. It should help users of this technology to define new Decision Diagram categories thanks to a simple specification mechanism called *Controller*. This enables the building of efficient Decision Diagrams dedicated to a given problem.

I. Introduction

Decision Diagrams (DDs) are now widely used in model checking as extremely compact representations of state spaces [1]. Numerous DD categories have been developed over the past twenty years based on the same principle (a brief survey is proposed in Section II).

DD categories now form a family of *abstract data types*, not only *data structures* [2]: each member of this family is designed to functionally manipulate specific mathematical structures such as Boolean Logic for Binary Decision Diagrams (BDDs) [3] and Zero-suppressed Decision Diagrams (ZDDs) [4], vectors in a finite subset of natural numbers for Multi-valued Decision Diagrams (MDDs) [5], term set rewriting for Σ Decision Diagrams (ΣDDs) [6], etc. They are all based on compact data structures associated with operations manipulating them.

However, each new DD category leads to a new definition. Thus, there is little sharing of principles, implementation and concepts between different ones. For instance, some DD categories define optimizations that cannot be extended to others due to a lack of generality in the definitions.

Another consequence is that users of DDs usually try to adapt their problem to existing categories due to the difficulty of providing a new one. A specification mechanism, that allows model checker designers to create problem-specific DDs, would be of interest.

The objective of this paper is to present polyDDs, a first step towards the definition of such a general DD Framework. We aim to group a large set of many currently existing DD categories, and to provide a simple specification mechanism with the notion of *controller*. A controller describes a DD *type*, *i.e.* a DD category (BDD, DDD, etc.) and a variable order (linear or not). By leaving variables and their domains as parameters, a whole DD category can be defined.

Our purpose is to introduce flexibility in the definition of DDs to adapt polyDD and have them behave like existing DD categories such as BDDs, MDDs or ΣDDs. Moreover, polyDDs can be used to define very specific DD-based representation for a given class of systems or even for a given specification. To our knowledge, this is the first attempt to exhort the user to define its own DDs.

This work is a first step in the direction of providing a parametric data type with the main specificities and power of DDs. On the one hand, some DD characteristics are not covered yet to focus on the basis, on the other hand, polyDD handles specifications that the existing DD categories do not. This structure is to be used jointly with user defined operations to implement efficient model checking algorithms.

The paper is structured as follows. Section II provides a brief survey of the main DDs characteristics, and specifies which ones are included or excluded from polyDDs. Section III provides both intuitive and formal definitions of the controller. Then, Section IV defines polyDDs. Finally, Section V defines, as an example, a DD category adapted to a specific real world problem from the robotic domain: CKBot configurations [7].

II. Decision Diagram Characteristics

All DD data structures share the same basic principle: each data to be encoded is represented as a path where nodes are labeled with variables and output arcs carry a value for the variable. The terminal node (also called terminal for short) is usually, but not always, associated with a value.

Then, a set of data is a set of paths where the common beginning parts are shared (*i.e.* represented only once), leading to the construction of a tree. Of course, only paths with *consistent* beginning parts can be put in the same set: if the first encountered difference between two paths is a node label, the two nodes cannot be shared and the paths cannot belong to the same set. The first encountered difference between two paths can, thus, only be a value.

Table I
CHARACTERISTICS OF SOME DDS

	Name	V.	O.	D.	Red.	Term.	H.	A.
(a)	BDD [3]	f	l.	$\mathbb{B}$	DC	$\mathbb{B}$	no	term.
	ZDD [4]	f.	l.	$\mathbb{B}$	ZS	$\mathbb{B}$	no	term.
(b)	ADD [8]	f.	l.	$\mathbb{B}$	DC	$\mathbb{N}$	no	term.
	MDD [5]	f.	l.	$\subset \mathbb{N}$	DC	$\mathbb{B}$	no	term.
	eMDD [9]	f.	l.	$\subset \mathbb{N}$	DC,ZS,Id	$\mathbb{B}$	no	term.
	IDD [10]	f.	l.	$\mathbb{R}^+$	DC	$\mathbb{N}$	no	term.
(c)	DDD [11]	n.f.	no	$\mathbb{N}$	no	$\mathbb{B}$	no	term.
	SDD [12]	n.f.	no	uns.	no	$\mathbb{B}$	yes	term.
	ΣDD [6]	n.f.	n.l.	T_Σ	no	$\mathbb{B}$	yes	term.
(d)	EVBDD [13]	f.	l.	$\mathbb{B}$	no	$\{0\}$	no	$+$
	EVMDD [14]	f.	l.	$\subset \mathbb{N}$	no	$\{0\}$	no	$+$
	WDD [15]	f.	l.	$\subset \mathbb{N}$	no	$\{0\}$	no	param.

Column names: V. (set of variables), O. (variable order), D. (variables' domain), Red. (reduction rule), Term. (number of values associated with the terminal), H. (hierarchy), A. (value associated with a path)
Abbreviations: f. (fixed), n.f. (not fixed), l. (linear), n.l. (not linear), uns. (unspecified), DC (Don't Care), ZS (Zero-suppressed), Id (Identity-suppressed), term. (the value is the terminal one), $+$ (the value is computed by adding values along the path), param. (the computation is a user-defined parameter).

To be more efficient, common bottom parts of the tree are also shared. The tree is then said to be a *reduced* diagram and forms a Directed Acyclic Graph (DAG), preserving the canonical representation. If more than one tree is stored, common bottom parts of all the trees are also shared.

BDDs [3] were the first DDs and were initally defined to store Boolean functions over a fixed number of Boolean variables. To share larger parts of the tree and to ensure consistency between paths, BDDs variables are met in the same order along all the trees paths. Such an order is said to be linear in the sequel. The DAG is then said to be an *ordered* diagram. As usual, when we refer to BDDs, we assume they are Reduced Ordered Binary Decision Diagrams (ROBDDs).

The variable ordering is also used, together with the variables domain, to further reduce the DAG: a predefined pattern can be safely removed from it when it can always be unambiguously restored from the variable order. Two such reduction rules have been introduced:

- "Don't Care": do not store a variable the value of which is meaningless (ROBDDs [3]),
- "Zero-Suppressed": do not store a variable if only *false* satisfies the function (ZDDs [4]).

BDDs have been a success from both an academic and an industrial point of view. So, they spread in various application areas: many DD-based structures appeared to extend their efficiency and their expressiveness. Some of them are summarized in Table I (a complete survey is out of scope of this paper).

The domain is the most obvious characteristic to be extended: MDDs are defined over finite subsets of $\mathbb{N}$, Data Decision Diagrams (DDDs) over $\mathbb{N}$, Interval Decision Diagrams (IDDs) over $\mathbb{R}^+$ and for Set Decision Diagrams (SDDs) the domain definition is user defined. This domain extension has lead to a new reduction rule ("Identity Suppressed") for the extensible MDDs (eMDDs): a node with a unique output arc labeled by a value v cannot have input arcs labeled by v.

Many DD-based structures inherit a fixed number of variables and a linear order from the BDDs. DDDs [11], SDDs [12] and ΣDDs [6] are examples of DDs with an unspecified number of variables: this way dynamic structures can be handled (lists, heaps, dynamic arrays, terms…).

Moreover, from the definitions of DDDs and SDDs, variables are not ordered and, thus, no reversible reduction rules can be defined. It limits the available operations. ΣDDs add constraints that define a non-linear (partial) order (variables depend on previously encountered values).

However, from a practical point of view, even DDDs and SDDs use implicit non-linear orders to avoid, e.g., binary operations (like union) on inconsistent paths. Controllers introduced in Section III can describe such non linear orders, allowing to generalize the reduction rule, that we name *Transparent Domains*, for those DDs.

When libraries were designed for DDs with linear order, a technical optimization was introduced: parts of the DAG leading to terminal 0 alone were not represented (thus a missing value on outgoing arcs from any node means that 0 is the leaf that will be encountered whatever the path followed afterward may be).

When DDs with unbounded variable domains were introduced (DDDs, SDDs and ΣDD), this technical optimization became part of the definition: infinite parts of the graph were linked to terminal 0 (and were not stored) in order to obtain finite structures. This feature is generalized as the *Vanishing Terminals* reduction rule in Sections III-C and IV-B.

Most DD categories are not hierarchical: they only share paths at the beginning and the end of the structure. Hierarchy, as in SDDs and ΣDDs, allows to recursively use DAGs to label arcs instead of values. This way parts that would be in the middle of a "flat" version of the DAGs (with few sharing possibilities) can be shared in a hierarchical version. Controllers in Section III define a general framework to handle hierarchy.

Finally, a value is associated with each path in the DAG. This way total functions can be stored using DDs. This value is, most of the time, the value associated with the terminal. But it can also be the result of a computation carried along the path. This is the case for Edge-Valued Binary Decision Diagrams (EVBDDs) [13], Edge-Valued Multi-Valued Decision Diagrams (EVMDDs) [14] and Weighted Decision Diagrams (WDDs) [15]. We do *not* handle such DDs.

For simplicity, we also only handle Boolean results, stored in terminals. Extension to other result types may seem obvious, as for Multi-Terminal Binary Decision Diagrams (MTBDDs) [16] a.k.a. Algebraic Decision Diagrams (ADDs) [8], but hierarchy introduces some subtleties that are left as future work.

III. CONTROLLER : PARAMETERIZING THE POLYDD FRAMEWORK

This section defines the *controller*. It describes a DD *type* by defining both a DD category (BDD, DDD, etc.) and a variable order (linear or not).

We follow the same approach as the description of words with a Deterministic Finite Automaton (DFA). So, a controller is an annotated graph that defines the shape of the DDs. It provides information about the DD characteristics: variables, associated domains, reductions, etc. Associated with a clearly defined semantics, it recognizes its compatible DDs. Moreover, reduction techniques and operations are defined according to the controller.

A. *Intuitive Introduction to Controllers*

To intuitively introduce controllers, we will now present how to describe some DD types, through an analogy with the ML type system: records (cartesian products) and unions.

1) Functional Type: Let us consider total functions `f` from Boolean variables (x,y,z) to one Boolean result:

```
type f = input -> output
```

where the domain and co-domain are defined as:

```
type input  = { x:bool ; y:bool ; z: bool }
type output = bool
```

Each of these functions can be represented by a BDD. One of them is shown in Figure 1, with variable order $x < y < z$.

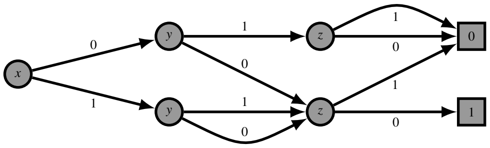


Figure 1. A BDD representing one function of **type** `f`

The controller for these functions, and for the BDD of Figure 1, is presented in Figure 2. Edge labels are triplets ⟨variable, variable domain, successor variable⟩. The chaining of variables referenced in edges labels defines the order. Although they seem redundant, the two variables are useful in more complex controllers, seen in the remainder of this section. The initial vertex is given by the incoming arrow, labeled with the initial variable. Variable τ denotes termination. A controller vertex together with a variable (initial or returned by an input arc) is called a *root*.

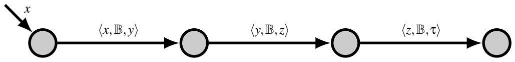
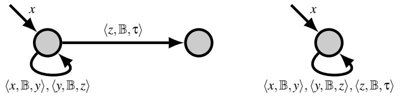


Figure 2. Controller for BDDs representing `f` with variable order $x < y < z$

The DD of Figure 1 is compatible with the controller of Figure 2. Compatibility can be intuitively explained as below. Every path in the BDD is also found in the controller. A controller path is for instance $x \xrightarrow{0} y \xrightarrow{1} z \xrightarrow{0} \tau$. Node variables in the DD are input variables on controller edges. Arcs are unfoldings of the domains on edge labels of the associated controller. So, arcs from a DD node cover the whole domain of their controller edge. Controller vertex with special variable τ corresponds to DD terminals 0 or 1.

2) Equivalent Controllers: The controller of Figure 2 is specific to a DD type, as it contains one vertex per variable. Controllers for other types in the same BDD category require more or fewer vertices, when the number of variables differs. For modeling and genericity purposes, the number of vertices in controllers for a whole category should remain constant, whatever the number of variables in the system is. Moreover, we require that at most one edge exists between two vertices. In order to share the same graph structure (same vertices and edges, but not same labels) between controllers of the same DD category, the idea is to define a controller pattern by folding the controller. Edge labels are then sets of triplets.

The controller of Figure 2 can be folded as in Figure 3. The loop edge defines the succession of variables in the described BDDs. So, only this edge label has to be rewritten according to the number of variables. It is a first step towards controller parameterization. Figure 3(a) shows a terminal vertex for variable z partially folded, whereas Figure 3(b) shows it as fully folded. Note that all these controllers are equivalent ($\equiv$): $C_1 \equiv C_2 \Leftrightarrow \mathit{Path}(C_1) = \mathit{Path}(C_2)$. Every DD compatible with one controller is compatible with the equivalent ones.



(a) Partially folded (b) Fully folded

Figure 3. Folded controllers from the one of Figure 2

3) Union Type: Let us consider the following type:

```
type t = A of { x1:bool ; y1:bool }
       | B of { x2:bool ; y2:int  }
```

This type hides a variable for the discriminant, that we call `t`, as its enclosing type. From the value of the `t` discriminant (A or B), two total variable orders are defined: $x_1 < y_1$ or $x_2 < y_2$.

Figure 4 shows the controller representing **type** `t`. The value of the discriminant (A or B) changes the variables on the following arcs. The variable domains differ for y_1 and y_2 even though they are on the same edge.

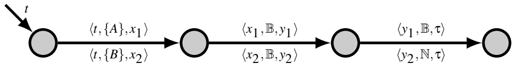


Figure 4. Controller for a union type in a BDD way

In most DD categories, the domains of y_1 and y_2 would raise a problem, as only one domain can be set for each edge. Two solutions exist. The first one is to create longer paths. It requires a controller with six vertices (one for each variable plus τ) as if the type was a tuple, like the controller in Figure 2. The second one extends the domain of y_1 to the largest one on the edge: $\mathbb{N}$, the domain of y_2. In both cases, some DD nodes and values are in fact useless. The "Don't Care" reduction rule is defined to remove them.

Let us now consider the following type:

```
type u = A of { z:bool }
       | B of { x:bool ; y:bool }
```

It is impossible to encode **type** `u` in the way we did for `t`. In fact, this is a limitation of the DD categories where all the paths have the same length.

Figure 5 presents the controller for **type** `u`. The value of the discriminant (A or B) changes the length of remaining paths, thus introducing more flexibility in the DD encoding of data. This encoding strategy is inspired from the one of DDDs [11].

4) Recursive Types: Union handling enables the representation of recursive types, such as lists. Terminal case (`Nil`)

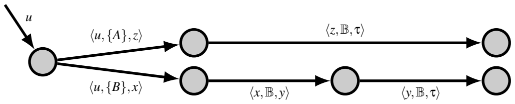

u
$\langle u,\{A\},z\rangle$
$\langle z,\mathbb{B},\tau\rangle$
$\langle u,\{B\},x\rangle$
$\langle x,\mathbb{B},y\rangle$
$\langle y,\mathbb{B},\tau\rangle$

Figure 5. Controller for a union type in a DDD way

is distinguished from the recursive one (`Cons`) to ensure finite recursions.

```
type l = Nil
       | Cons of { x:bool ; n:l }
```

Figure 6 shows the controller for **type** `l`. A cycle is mandatory to represent type recursion, as an unfolded controller would be infinite. Introduced by DDDs, it allows to encode unbounded data structures such as lists or FIFOs. This kind of DD still represents finite structures, unbounded means here that we cannot fix the bound in advance. Of course, truly infinite structures cannot be represented, as in most programming languages.

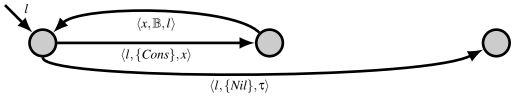

l
$\langle x,\mathbb{B},l\rangle$
$\langle l,\{Cons\},x\rangle$
$\langle l,\{Nil\},\tau\rangle$

Figure 6. Controller for a recursive type

The DD of Figure 7 is compatible with the controller of Figure 6. As the represented functions are total, every DD of a recursive controller is infinite: here lists of unbounded length are represented. A reduction, called *Vanishing Representatives*, presented in section IV-B1, removes some nodes and edges so as to leave only a finite part. The node labeled by "…" represents the infinite part of this DD, where all paths lead to terminal value 0, as for DDDs. The following lists are the only data for which the represented function returns 1:
`[ 1 ], [ 1; 1 ], [ 1; 0 ], [ 0; 0 ], [ 0 ;1 ]`

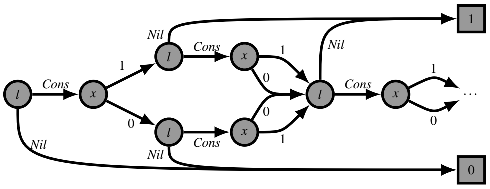


Figure 7. A DD compatible with the controller in Figure 6 (the representation of infinite paths, here with "…", is seen in Section IV-B1)

5) Hierarchical Representation: Sharing amount in DDs depends on the variable order, but finding the optimal order is NP-Complete [17], [18], [19]. In practice, when the paths lengths increase, finding a good variable order is harder and sharing decreases. Some DDs such as SDD [12] or ΣDD [6] allow hierarchical encoding of complex structures in order to increase the sharing of common parts. Hierarchy requires ternary edges in controllers, and is represented by the —(arrow tip.

Figure 8 represents a hierarchical controller associated with **type** `l`. It is composed of two parts: the list with a recognizable loop, and the list content (a *Cons* and a Boolean). The hierarchical edge is labeled by a triplet of variables. The first and last one have the same meaning as for flat edges, whereas the second variable is the initial one given to the hierarchy, here c. It means that only DDs compatible with the corresponding root can label the hierarchical DDs arcs.

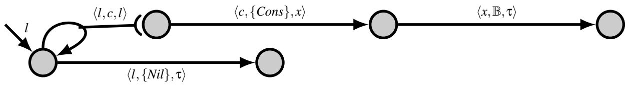

l
$\langle l,c,l\rangle$
$\langle c,\{Cons\},x\rangle$
$\langle x,\mathbb{B},\tau\rangle$
$\langle l,\{Nil\},\tau\rangle$

Figure 8. Controller for a hierarchical encoding of `l` à la SDD

The encoding in Figure 8 has two levels of hierarchy: a level for the list with unbounded path length, and a level for its stored data with a fixed path length. Figure 9 shows an alternate hierarchical encoding of **type** `l`, closer to the one proposed by ΣDDs in [6]. It contains only one level, with a recursive hierarchy. Here paths lengths are all fixed, but hierarchy depth is unbounded.

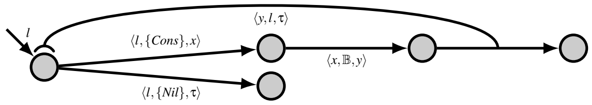

l
$\langle y,l,\tau\rangle$
$\langle l,\{Cons\},x\rangle$
$\langle x,\mathbb{B},y\rangle$
$\langle l,\{Nil\},\tau\rangle$

Figure 9. Controller for a hierarchical encoding of `l` à la ΣDD

6) Generalizing Edge Labels: The current representation of edge labels is not satisfactory since:

- it may describe non deterministic controllers which prevents a sound definition of reductions and operations;
- it may lead to long enumerations, difficult to handle.

To be deterministic, the triplets on an edge label must correspond to a partial function f : variable × value → variable for flat edges or f : variable × variable → variable for hierarchical ones. In the graphical representation, these functions are defined *extensionally* (as with triplets) or *intentionally* (with functions, using $\mapsto$).

As variables and their domains are not enumerated, they can be left as parameters. A controller with an intentional description can thus define a whole category, not only a type.

The controller in Figure 10 represents a list of increasing domains. As there is no bound for the list depth, it would require an infinite number of triplets on the edge labels. Intentional description is more powerful than enumeration. The formalization in Section III-B, which is based on an extensional description, does not depend on the finiteness of edge labels. Thus it handles intentional descriptions too.

To our knowledge, no existing DD category can precisely handle the DDs of the controller in Figure 10. They either require a bound on variable domains as in MDDs, or unbounded ones as in DDDs. It shows part of the value added by polyDDs.

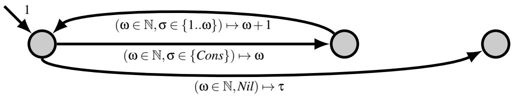

1
$(\omega \in \mathbb{N}, \sigma \in \{1..\omega\}) \mapsto \omega + 1$
$(\omega \in \mathbb{N}, \sigma \in \{Cons\}) \mapsto \omega$
$(\omega \in \mathbb{N}, Nil) \mapsto \tau$

Figure 10. Controller for a parameterized list of increasing domains. The variables are integers. The domain associated with a variable $\omega \in \mathbb{N}$ is $\{1 \ldots \omega\}$. Starting from variable 1, there is no predefined bound to the recursion depth.

B. Controller definition

A controller $C = \langle \Omega, \Sigma, V, E, I \rangle$ is a graph with binary (flat) or ternary (hierarchical) labeled edges. We give its formal definition in the remainder of this section.

1) Variables (Ω): The set of variables is Ω. It is a non empty discrete set, but can be infinite. It contains at least the special terminal variable, $\tau \in \Omega$. We denote $\omega, \omega_i, \ldots$ its variables.

2) Values (Σ): The set of edge values is Σ. It can be finite or infinite, discrete or continuous, but must be non empty. We denote $\sigma, \sigma_i, \ldots$ values in this set.

3) Vertices (V): The set of controller vertices is V. It is non-empty and *finite*. A root is composed of two parts: a controller vertex and its initial variable. The set of possible roots is thus defined as $R = V \times \Omega$.

4) Initial roots (I): A subset $I \subseteq R$ of the roots is chosen as the initial roots of the controller.

5) Edges (E): There are two kinds of edges in a controller: flat ones and hierarchical ones. We replace the usual use of vertices in graph edges definition by roots. Flat edges are given by the set $E_f \subseteq R \times \Sigma \times R$ and hierarchical edges by the set $E_h \subseteq R \times R \times R$. The set of edges is $E = E_f \cup E_h$.

6) Determinism: A controller is deterministic. This property is given by predicate $det(C)$. From each vertex, all edges have disjoint values or hierarchical roots.

$$det(C) \Leftrightarrow \forall r \in R, \forall x \in \Sigma \cup R, |\{r' \mid \langle r, x, r' \rangle \in E\}| \leq 1$$

Labels in graphical representation are partial functions of type $\Omega \times \Sigma \to \Omega$ for flat edges, or $\Omega \times \Omega \to \Omega$ for hierarchical ones. Edges are relations in their formal definition, but the required determinism ensures that they are functions.

For a given controller C, the set of paths $Path(C)$ collects all sequences of variables and values leading to a terminal that appear on controller edges.

When following a path, some clearly useless edges, like $\langle \tau, \mathbb{B}, x \rangle$ are permitted by this definition. Even if they cannot be followed, these edges are *not* an error. Sometimes, they are useful to model compactly a controller. Consistency rules given in Section IV-A3 remove these useless paths.

C. Reductions definition

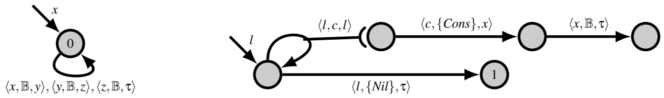
Two kinds of reductions are usually defined for DDs. The first one (*vanishing terminals*) removes a default terminal value. The second one (*transparent domains*) removes edges labeled by a default set of values. Whereas most DD categories define reversible reductions, some (like DDDs) do not and thus lack some operations. In polyDDs, the reductions are defined to be reversible. Each one requires some annotations on the controller. Thus, we enhance the controller definition with two annotations Φ and Ψ that define the reductions to apply, so now a controller with reduction is: $C = \langle \Omega, \Sigma, V, E, I, \Phi, \Psi \rangle$.

1) Vanishing terminals (Φ): The vanishing terminals reduction has first been defined for BDDs. It has spread to many other categories. Vertices such that all paths lead to a default terminal value (usually 0) are removed from the DDs. Edges leading to these vertices or to the default terminal are also removed. This reduction is optional for most of the DD categories, but mandatory for DDDs and derived: each of these DDs would be infinite or non-deterministic otherwise.

Annotations on controller vertices give the default terminal values, by attaching a Boolean value (or nothing) to any vertex. The total function $\Phi : V \to \mathbb{B} \cup \{\bot\}$ returns the vanishing terminal value for each vertex, or the value $\bot$ if none is defined. In fact, vanishing annotations are only useful for vertices which can take τ as input variable. To be valid, all the paths through controller cycles must lead to a vanishing terminal. This constraint is formalized in Section IV-B1.

Figure 11 shows two controllers with vanishing terminal annotations: 11(a) for a controller of BDDs and 11(b) for a controller of lists. The vanishing terminal annotation is given inside the vertices ($\bot$ is not represented). In Figure 11(a), all paths leading to terminal value 0 disappear in the BDDs.

Figure 11(b) is a controller for lists. Given an infinite set L of all lists, partitioned in two subsets L_1 and L_2, a DD representation is possible only when L_1 or L_2 is finite. All the DD paths for the infinite part must lead to the vanishing terminal, whereas the paths for the finite part have no constraint. Notice that we do *not* require 0 to be the vanishing terminal: it is chosen by the user. In the figure, the controller is adapted when representing an infinite number of lists, with a finite complement. All terminal vertices do not require a vanishing terminal value. For instance, We specify no vanishing terminal in the hierarchy.


x
0
$\langle x, \mathbb{B}, y \rangle, \langle y, \mathbb{B}, z \rangle, \langle z, \mathbb{B}, \tau \rangle$
l
$\langle l, c, l \rangle$
$\langle c, \{Cons\}, x \rangle$
$\langle x, \mathbb{B}, \tau \rangle$
1
$\langle l, \{Nil\}, \tau \rangle$

(a) From Figure 3(b)

(b) From Figure 8 (the DDs will represent the finite complement of an infinite number of lists)

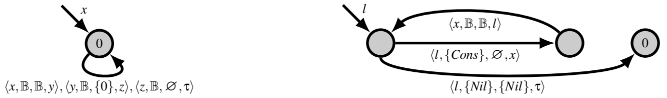
Figure 11. Vanishing terminal annotations for two controllers

2) Transparent domains (Ψ): The transparent domains reduction is found in several DD categories, under several names. For instance, BDDs use the “Don’t Care” rule, whereas ZDDs use the “Zero-Suppressed” one. We generalize these rules in the transparent domains reduction.

This reduction removes DD nodes if they have only one successor, that can be reached uniquely through a default domain. It requires optional annotations on controller edges to define the default domains. For flat edges, a subset of the edge domain is given. For hierarchical ones, it is replaced by a default DD (which represents a default domain, too).

We do not describe the hierarchical transparent domains here, as DDs are defined later, but there is no dog chasing its tail as the reduction is *not* mandatory to build DDs. As for vanishing terminals, transparent domains are given by a set: $\Psi \subseteq R \times \Sigma \times R$. We do not formally require the transparent domain to be a subset of the edge domain. However, other cases do not enable the reduction and are thus useless.

Graphically, we add another domain to the edge labels where this reduction is defined (after the real edge domain, before the output variable). By doing so, we handle cases where $\Psi \subseteq E$. They are sufficient to define useful transparent domains, and are readable. In this article, we use triplets on controller edges when transparent domains are not relevant, and quadruplets otherwise. Figure 12 shows these annotations on already seen controllers.



(a) From Figure 3(b) (b) From Figure 6

Figure 12. Transparent domains for two controllers

The transparent domains ($\mathbb{T}_i$) for a given controller C must respect one condition, $tr(C)$. There must be at least one empty transparent domain in each cycle to be able to count the cycles. Otherwise, the reduction is not reversible.

$$tr(C) \Leftrightarrow \forall r_1 \xrightarrow{\mathbb{D}_1,\mathbb{T}_1} \ldots \rightarrow r_n \xrightarrow{\mathbb{D}_n,\mathbb{T}_n} r_1 \in Path(C), \exists i \in \{1 \ldots n\}, \mathbb{T}_i = \varnothing$$

For example, the controller in Figure 13 does not follow this rule. It contains a cycle labeled by $\langle l, \{Cons\}, \{Cons\}, x\rangle$ and $\langle x, \mathbb{B}, \mathbb{B}, l\rangle$ that contains no $\varnothing$ transparent domain.

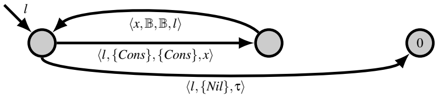


Figure 13. Erroneous transparent domain annotations

IV. polyDD: Polymorphic Decision Diagrams

A controller describes the DDs of a given type. The relationship between the controller and its compatible DDs in our approach can be compared to words and their recognition by a DFA. A controller recognizes DDs, which are not linear sequences of letters like words but hierarchical graphs. The recognized language is thus the set of compatible DDs.

polyDDs are hierarchical DDs with parameterized variable domains and Boolean terminals. They share a lot with SDDs and ΣDDs that are the only hierarchical DDs. But the complement (vanishing part) of a DDD or an SDD cannot be represented as a DD (see [11]). In polyDDs, as the controller is deterministic, we can define the vanishing part, even if it is infinite. Because of this potential infiniteness, we do not provide an inductive definition as for SDDs. We rather define a polyDD as the result of graph transformations on its controller. When applying them, we progressively reach one of the DDs compatible *by construction* with the controller.

DDs are acyclic subgraphs of their controller. Moreover, their arcs are labeled only by a value or a domain, and their terminals are given a Boolean value.

First, the controller is unfolded to create the "shape" of a DD (Section IV-A). Then, it is transformed to a real DD in Section IV-B, by removal of its remaining cycles and binding of the terminals. Then hierarchical parts are defined and identical nodes are merged to enable sharing. Finally, reductions are applied in Section IV-C to optimize the DD.

A. Unfolding

From user defined controllers, unfolding generates a set of possible equivalent controllers (in the sense of Section III-A). An unfolding builds a pattern of Decision Trees, reduced later into a Decision Diagram in Section IV-B5. It is composed of two transformations: vertex splittings (Section IV-A1) then edge splittings (Section IV-A2). Each one is applied a bounded number of times, *not* necessarily until a fixpoint is reached. A consistency rule applied between all transformations removes dead paths (Section IV-A3). When a special condition on unfolding is met (Section IV-A4), DD generation *can* continue to the next steps (Section IV-B), but also do more unfoldings.

1) Vertex splitting: This first transformation creates vertices with only one incoming edge. It is mainly used to unfold controller cycles, but is also useful for edge splitting, as it requires that the destination vertex has only one incoming edge.

Transformation rule 1 (Vertex splitting)**.** *A vertex with incoming edges or initial variables can be split in several vertices. The transformed vertex is removed.*

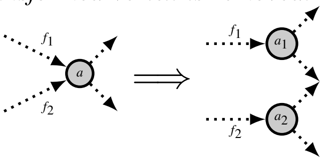


Figure 14 shows a vertex splitting on the controller for lists of Figure 6. The initial root (leftmost vertex) is split. The first created vertex takes the initial root as input, whereas the second created vertex takes $\langle x, \mathbb{B}, l\rangle$. Output edges are copied for both vertices.

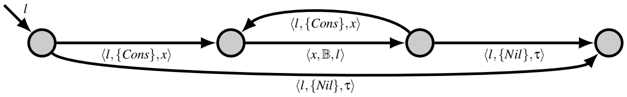
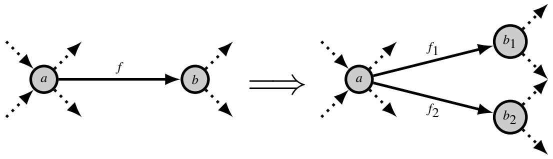


Figure 14. Result of a vertex splitting on the controller of Figure 6

2) Edge splitting: This transformation splits an edge in two parts. By doing so, it creates separate paths, that can be later bound to different terminal values.

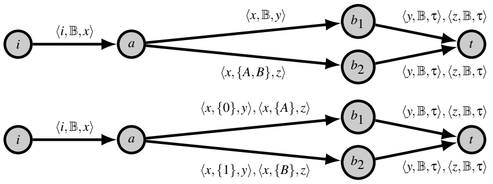
Transformation rule 2 (Edge splitting)**.** *An edge can be split in two parts, if and only if it is the sole incoming edge of its destination vertex. The destination vertex is duplicated and edges are created from the source vertex to the newly created ones. The original edge and the original destination vertex are removed. Labels on new edges differ between flat and hierarchical edges:*

- *for flat edges, the function on original edge is partitioned in two parts ($f = f_1 \cup f_2 \wedge f_1 \cap f_2 = \varnothing$),*
- *for hierarchical edges, the function is kept the same for the two new edges ($f = f_1 = f_2$).*

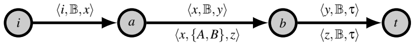


A controller edge can be split in several ways, and the resulting controller can usually be unfolded again. Figure 15 shows two different edge splittings of a controller. Notice that each one can be split again.

Hierarchical edge splitting differs from flat edge splitting, because the hierarchical link is not a domain that can be split. It represents a domain, that is later instanciated by a DD. A condition given in Section IV-B4 ensures later that hierarchical domains on hierarchical DD arcs are disjoint.



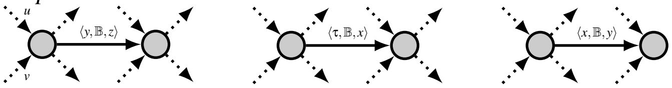
i
$\langle i,\mathbb{B},x\rangle$
a
$\langle x,\mathbb{B},y\rangle$
$\langle x,\{A,B\},z\rangle$
b_1
b_2
$\langle y,\mathbb{B},\tau\rangle,\langle z,\mathbb{B},\tau\rangle$
$\langle y,\mathbb{B},\tau\rangle,\langle z,\mathbb{B},\tau\rangle$
t
i
$\langle i,\mathbb{B},x\rangle$
a
$\langle x,\{0\},y\rangle,\langle x,\{A\},z\rangle$
$\langle x,\{1\},y\rangle,\langle x,\{B\},z\rangle$
b_1
b_2
$\langle y,\mathbb{B},\tau\rangle,\langle z,\mathbb{B},\tau\rangle$
$\langle y,\mathbb{B},\tau\rangle,\langle z,\mathbb{B},\tau\rangle$
t

Figure 15. Two different edge splittings (with named vertices, not vanishing annotations) for 

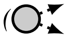
i
$\langle i,\mathbb{B},x\rangle$
a
$\langle x,\mathbb{B},y\rangle$
$\langle x,\{A,B\},z\rangle$
b
$\langle y,\mathbb{B},\tau\rangle$
$\langle z,\mathbb{B},\tau\rangle$
t

3) Path consistency: Transformation rules may generate controllers with dead paths, where the special variable τ cannot be reached. They are dead paths in the initial controller too, but are usually handy to define the controller. Applied iteratively, the consistency rules given below remove all dead paths.

Transformation rule 3 (Inconsistent paths)**.** *Edges that match one of the patterns below are removed. Notice that the incoming edges are either an initial root or an edge from another vertex. The label shown in the patterns is then its output variable.*



u
v
$\langle y,\mathbb{B},z\rangle$
$\langle \tau,\mathbb{B},x\rangle$
$\langle x,\mathbb{B},y\rangle$

Unreachable nodes ( *pattern) are also removed.*

After all the unfolding transformations, we get a flat-deterministic controller where all finite paths end with the special variable τ and start with an initial variable. All along the paths, output variables can be taken as input by the successor vertices. Some cycles may remain in the controller, even with no reachable ending variable τ. They are removed in Section IV-B when possible.

Theorem 1 (Flat-determinism preservation)**.** *From a controller which is deterministic for flat edges, unfolding generates equivalent flat-deterministic controllers.*

$$unfold(C) = \{C_1 \ldots C_n\} \ s.t. \ \forall C_i, C_i \equiv C$$

Proof Idea: *No new outgoing edges are added to existing vertices by splitting. Outgoing edges added to an existing vertex by unfolding are disjoint.* □

Of course, vanishing terminal and transparent domain annotations must be taken in account in unfolding. When a vertex is duplicated, its vanishing terminal annotation is copied to each created vertex. When an edge is split, so are the transparent domain annotations on it.

4) Roots uniqueness: A controller vertex can take several variables as input, whereas a DD node has exactly one variable. Thus, we define a property on controllers: a controller C has unique roots (predicate $ru(C)$) if and only if each vertex is associated with only one variable. Such a controller may be transformed to DDs (Section IV-B gives other constraints).

$$ru(C) \Leftrightarrow \forall i,o \in V, |\{v_i \mid \langle\langle i,v_i\rangle, x, \langle o,v_o\rangle\rangle \in E\}| \leq 1$$

B. From controller to polyDD

We define polyDDs from unfoldings of a controller, starting with an initial root. A polyDD is a DAG $D = \langle C,i,N,A,M\rangle$, where C is its unfolded controller, $i \in I$ one of its initial roots, N is the set of labeled nodes, $A = A_f \cup A_h$ the set of labeled arcs and $M : N \to R$ is a mapping from nodes to controller roots. As in controllers, arcs are flat ($A_f \subseteq N \times \Sigma \times N$) or hierarchical ($A_h \subseteq N \times N \times N$). Once a controller C with the DD shape is obtained from the user controller C_u, $C \in Unfold(C_u)$ (with usually less sharing than in the final DD), several steps are required. They are described in this section, in order.

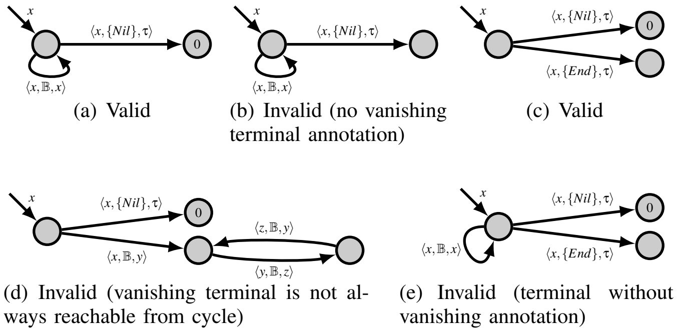
1) Vanishing representatives: This step removes the remaining controller cycles, to transform the graph into a DAG. To do so, we identify vertices belonging to a cycle. They are replaced by new vertices, called *vanishing representatives*, wthout successors.

A root is said vanishing-reachable (predicate $vr(r)$) if and only if a path exists from this root to a vertex ending with variable τ. Moreover, every terminal reachable from the vertex must be labeled by a vanishing terminal annotation.

$$\begin{aligned} vr(r_1) \Leftrightarrow\ & \exists r_1 \xrightarrow{\mathbb{D}_1,\mathbb{T}_1} \ldots \to r_n \xrightarrow{\mathbb{D}_n,\mathbb{T}_n} \langle o,\tau\rangle \in Path(C) \\ \wedge\ & \forall r_1 \xrightarrow{\mathbb{D}_1,\mathbb{T}_1} \ldots \to r_n \xrightarrow{\mathbb{D}_n,\mathbb{T}_n} \langle o,\tau\rangle \in Path(C), \Phi(o) \neq \bot \end{aligned}$$

All roots belonging to cycles in the unfolded controller (a vertex and its input or initial variable), must be vanishing-reachable. If a controller does not obey this rule, it is invalid and no compatible DDs exist.

Figure 16 shows several valid and invalid controllers. 16(a) is valid, as all the paths from the loop can reach the vanishing terminal 0. As this is not true in 16(b), the latter is invalid. 16(c) is valid, as the constraint only applies on cycles. Figures 16(d) and 16(e) show invalid controllers, because some paths through cycles do not lead to a vanishing terminal.



x
$\langle x,\{Nil\},\tau\rangle$
0
$\langle x,\mathbb{B},x\rangle$
x
$\langle x,\{Nil\},\tau\rangle$
$\langle x,\mathbb{B},x\rangle$
x
$\langle x,\{Nil\},\tau\rangle$
0
$\langle x,\{End\},\tau\rangle$
x
$\langle x,\{Nil\},\tau\rangle$
0
$\langle z,\mathbb{B},y\rangle$
$\langle x,\mathbb{B},y\rangle$
$\langle y,\mathbb{B},z\rangle$
x
$\langle x,\{Nil\},\tau\rangle$
0
$\langle x,\mathbb{B},x\rangle$
$\langle x,\{End\},\tau\rangle$

(a) Valid

(b) Invalid (no vanishing terminal annotation)

(c) Valid

(d) Invalid (vanishing terminal is not always reachable from cycle)

(e) Invalid (terminal without vanishing annotation)

Figure 16. Vanishing reachability for some controllers

Figure 17 shows a DD that would be compatible with the controller of Figure 16(e), if it were valid. Defaults parts that can be inferred from vanishing terminal annotations are dotted. This DD is still infinite, because there is no default terminal value for all paths through *End*. It represents instances of `type t = End | Nil | bool * t` that do not contain `Nil` (all the DD paths through *Nil* lead to terminal 0).

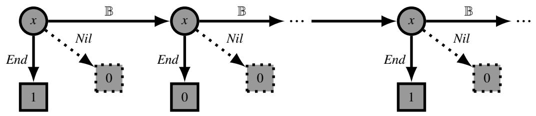


Figure 17. Infinite DD that cannot be made finite, compatible with the controller of Figure 16(e)

As vanishing reachability ensures that all cycles lead only to vanishing terminals, their vertices can be replaced by *vanishing representatives*. They are vertices with no successors. Each one represents the infinite vertex unfolding of the cycle where all paths lead to vanishing terminal values *only*. This transformation creates orphan vertices, that can be safely removed.

Figure 18 shows the vanishing representatives for an unfolding of the list controller given in Figure 6. Vanishing representatives are (here only) marked with double circles to distinguish them from other vertices. Each vanishing representative represents the empty set of lists. So, the DDs compatible with the controller represent lists of at most one element.

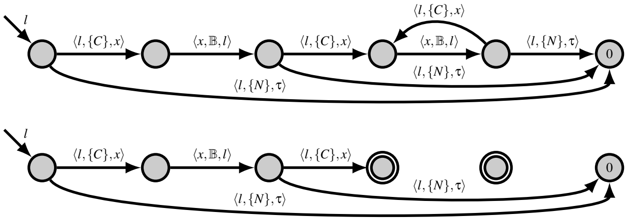

l
$\langle l,\{C\},x\rangle$
$\langle x,\mathbb{B},l\rangle$
$\langle l,\{C\},x\rangle$
$\langle l,\{C\},x\rangle$
$\langle x,\mathbb{B},l\rangle$
$\langle l,\{N\},\tau\rangle$
0
$\langle l,\{N\},\tau\rangle$
$\langle l,\{N\},\tau\rangle$
l
$\langle l,\{C\},x\rangle$
$\langle x,\mathbb{B},l\rangle$
$\langle l,\{C\},x\rangle$
0
$\langle l,\{N\},\tau\rangle$
$\langle l,\{N\},\tau\rangle$

Figure 18. Vanishing representatives for an unfolding of Figure 6 (*C* is used for *Cons* and *N* for *Nil* to get a compact figure)

2) Vertices and edges relabeling: After the vanishing representatives transformation, the controller is a DAG. We change its vertex and edge labels to be those of a DD. Each vertex is labeled by a root, and each edge by a domain.

Theorem 2 (Node to root mapping)**.** *Every DD node corresponds to exactly one controller root.*

Proof Idea: *Unfolding operations (Section IV-A) create new vertices by copying controller ones, or remove unused edges or vertices. Edge labels or initial roots are kept or split. DD transformations (Section IV-B) remove some edges. They also change labels, but this change is syntactical.* □

As there is now only one initial root, and the controller has unique roots (as stated in Section IV-A4), this transformation is not ambiguous. Because of these changes, we rename the vertices and edges: they are now called nodes and arcs, as in the usual DD terminology.

Figure 19 shows the relabeling of the controller in Figure 18. Vertex labels are taken from the input edges. We show here only the variable part of their labels (which are roots), as in usual DD representations. Edge labels are the domains on the controller edges. The orphan node is shown but can be removed (and is removed in the remaining sections).

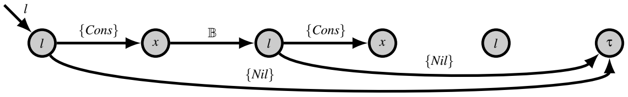


Figure 19. Vertex and edge relabeling for the controller of Figure 18

3) Terminal binding: The next step is to set a terminal Boolean value for each node labeled by the terminal variable τ. Figure 20 shows the result of this transformation for the controller of Figure 19. Notice that we do some additional (and optional) edge splittings to get two distinct terminals. The vanishing terminal 0 is not yet removed. It is used until now only to remove cycles.

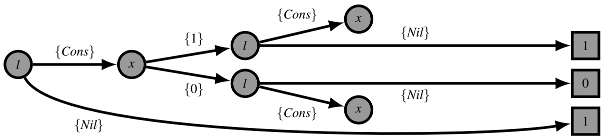


Figure 20. Terminal bindings based on the controller of Figure 19

4) Hierarchy: Hierarchical edges are instanciated: each hierarchical root is replaced by a DD compatible with it. There can be no cycle through the hierarchical arcs, as flat ones.

The canonicity constraints given in [12] apply: hierarchical DDs on the outgoing arcs of a node must be disjoint and their union must be the full DDs (where all paths lead to terminal 1). No empty DD (where all paths lead to terminal 0) is allowed to label an arc.

5) Unicity: As usual with DDs, each node must be *unique*. Thus, all nodes that are identical are merged. Two nodes are identical if and only if they have the same variable label, and have the same successor nodes for the same edge domains. When two nodes are merged, the labels on their input arcs from the same source node are also merged. The algorithm can again be found in [12] or other DD articles.

C. *Reductions*

Reductions, like removal of the 0 terminal and "Don't Care" or "Zero-Suppressed" are applied on the DD. In polyDD, we ensure their reversibility: the original structure can be recovered, knowing the reduced one and the controller. This is not the case in all DD categories that have been defined. For instance, even the (infinite) complement cannot be derived from a DDD because the missing parts are ambiguous. Moreover, DDDs (and derived) cannot define the transparent domains reduction, because of their lack of order specification.

1) Vanishing terminals: The vanishing terminal annotations are used in Section IV-B1 to remove cycles in the controller. But a lot of other categories with linear controllers, like BDDs, use them to reduce the DD size. Figure 21 shows a BDD controller (taken from Figure 3(b)) with vanishing terminal 0 (as usual for this category), and a DD compatible with this controller (taken from Figure 1) with dotted vanishing parts.

This rule is defined in three parts, applied recursively from the terminals to the root:

a) a terminal is vanishing if and only if it is labeled by the vanishing terminal annotation of its controller vertex or it is a vanishing representative;

b) an arc is vanishing if and only if it is labeled by the domain on its controller edge and leads to a vanishing node or terminal; on hierarchical arcs, the full domain is the DD where all paths lead to terminal 1;

c) a node is vanishing if and only if all its outgoing arcs are vanishing.

All vanishing terminals, arcs and nodes can be safely removed from the DD. As several terminal vertices can exist in the controller, each one can be associated with its vanishing value. To our knowledge, polyDD is the only DD category able to mix several vanishing terminals.

Theorem 3 (Vanishing terminals reversibility)**.** *The vanishing terminals reduction is reversible.*

Proof Idea: *The controller defines edge domains. Moreover, it is deterministic. As there is a mapping between DD nodes and controller vertices, this reduction is reversible: we can build the missing parts of the DD.* □

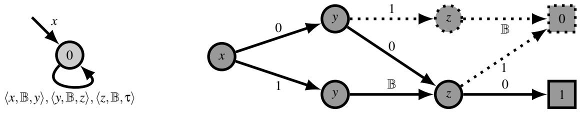


Figure 21. Example of vanishing terminals reduction

2) Transparent Domains: Transparent domains reduction removes some nodes in the DD when they have only one remaining successor, after the vanishing terminals reduction. If a node has only one successor and the arc domain to this successor is the transparent domain of the corresponding controller edge, then the node and its outgoing arc are removed, and all its input arcs are linked to its successor.

Figure 22 shows the steps of this reduction on the BDD of Figure 21. The transparent domains here depend on the variables. They can be adjusted by the user to enhance the DD compactness. For instance, we mix the BDD ($\mathbb{B}$) and ZDD ($\{0\}$) patterns.

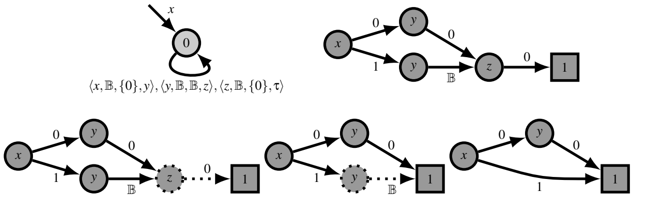


Figure 22. Example of transparent domains reduction (mixed BDD and ZDD patterns)

Theorem 4 (Transparent domains reversibility)**.** *The transparent domains reduction is reversible.*

Proof Idea: *When a cycle appears in the controller, the reversibility condition given in Section III-C2 ensures that the original DD can be recovered unambiguously. Without this restriction, we would not know how many occurrences of the cycle have disappeared. In acyclic controllers, this reduction is always reversible.* □

Recently, "identity" patterns have been proposed in [9]. These patterns are a special kind of transparent domains: the domain on a controller edge depends on the values seen on the following DD arcs. To be generalized, this pattern requires more complex transparent domain annotations. However, the reduction proposed in this article generalizes already most of the patterns found in DD categories.

D. Ensuring Canonicity

The most important property of DDs is canonicity. It states that, from a user specified controller and a data to encode, at most one DD exists for its representation. Some data cannot be represented.

Theorem 5 (Canonical representation)**.** *Given a controller, polyDD offer a canonical representation of data.*

Proof Idea: *As polyDDs keep most of their properties from SDDs [12], the proof given for them is almost valid. But we replace one of their specific rules (every node with arcs only to terminal* 0 *is itself equivalent to this terminal) with vanishing representatives. The vanishing terminals reduction gives an equivalent behavior to polyDDs. As reductions are reversible and not ambiguous, they do not break canonicity.* □

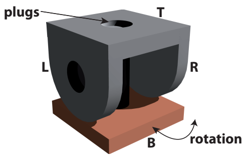
E. Manipulation

The DDs considered in this article represent total functions with Boolean results only. Binary operators $\cup, \cap, \setminus$ can be defined for polyDDs as they are for SDDs. They are internal operations *i.e.* union of DDs compatible with the same controller root is itself compatible with the same root.

As all reductions are reversible, the worst case is to build the non-reduced DDs and apply the operator to them. Algorithms for these operations in other DD categories, like ZDDs or MDDs, are optimized to benefit from the reductions. Because we generalize the usual reductions, most of these optimizations can be reused.

V. Specifying a Structure Dedicated to a Problem

Let us consider a modular robotic problem extracted from [7] (CK-Bot). A modular robotic system is composed of several modules having the same shape and deploying the same software. A module has three fixed sides (L, T, R) that can plug to another module. The fourth side (B) also has such a plug. Moreover, it can rotate from $-90°$ to $+90°$ around an axis. In this simplified model, no other rotation is defined.



We aim to enumerate all possible configurations for a system embedding N CKbots. For that purpose, we define a dedicated representation using dedicated DDs parameterized with a controller. To do so, we first model the system using types, as in Section III-A (warning: they are *not* valid ML syntax but close to it), and then produce the corresponding controller. The model checking part as well as a discussion on the controller quality are out of the scope of this paper.

A. Data types

As DDs heavily rely on dynamic programming and sharing, we define data types that enhance sharing. Storing module positions is thus clearly not a good idea. We rather define them from an origin module and encode the rotation of the mobile part and the position of other modules using the orientation of their top face. All modules build a connected graph.

We first define module identifiers: `type id = 1 .. N`.

Each module has an orientation in space (24 states: 6 for the top face orientation and 4 rotations around the top axis).

```
type orientation = ( PlusX  | PlusY  | PlusZ
                   | MinusX | MinusY | MinusZ )
                 * ( 0, 90, 180, 270 )
```

We need to encode the state of the rotating part:

```
type rotation = -90 | 0 | +90
```

The states of all modules are grouped in an array. Orientation of the origin module never changes.

```
type modules = { m1 : (PlusX, 0)  * rotation
               ; m2 : orientation * rotation
           ... ; mN : orientation * rotation }
```

As two unlinked modules can be side-by-side, we need to encode the plug state. This is done via a list of records storing the identifier of plugged modules:

```
type plug = { fst : id ; snd : id }
type plugs = Nil | { current : plug ; next : plugs }
```

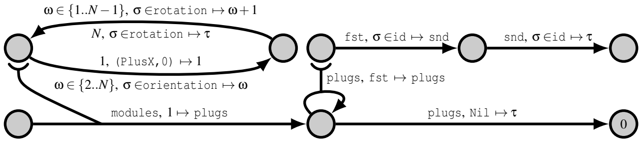
Module orientations are sufficient to retrieve the plugged sides so we do not need to store this information. As plugs are symmetric, we can of course sort them, and for instance always store the lowest identifier in `fst`. This is relevant for operations, not for the type description.

So, the full system is encoded with the following type:

```
type ckbots = { ms : modules ; ps : plugs }
```

B. Controller

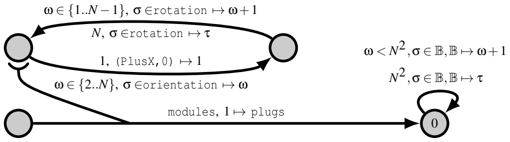
The controller of `ckbots` is presented in Figure 23. Its construction follows the rules defined for Figure 10. The edge labeled by `1,(PlusX,0)`↦`1` specifies a constraint that cannot be expressed using other DD categories. As there is a cycle for the plugs, a vanishing terminal is required.



$\omega \in \{1..N-1\}$, $\sigma \in$rotation $\mapsto \omega+1$
N, $\sigma \in$rotation $\mapsto \tau$
1, (PlusX,0) $\mapsto 1$
$\omega \in \{2..N\}$, $\sigma \in$orientation $\mapsto \omega$
modules, $1 \mapsto$ plugs
fst, $\sigma \in$id $\mapsto$ snd
snd, $\sigma \in$id $\mapsto \tau$
plugs, fst $\mapsto$ plugs
plugs, Nil $\mapsto \tau$
0

Figure 23. Controller for the CKBot problem (plugs list)

Figure 24 shows another way to encode `ckbots`. Here, the plug list is replaced by a Boolean $N \times N$ matrix. The vanishing terminal is no more required, but we keep it to reduce the DDs. The plugs matrix is expected to be very sparse, so we choose the BDD pattern ($\mathbb{B}$) for transparent domains.



$\omega \in \{1..N-1\}$, $\sigma \in$rotation $\mapsto \omega+1$
N, $\sigma \in$rotation $\mapsto \tau$
1, (PlusX,0) $\mapsto 1$
$\omega \in \{2..N\}$, $\sigma \in$orientation $\mapsto \omega$
modules, $1 \mapsto$ plugs
$\omega < N^2, \sigma \in \mathbb{B}, \mathbb{B} \mapsto \omega+1$
$N^2, \sigma \in \mathbb{B}, \mathbb{B} \mapsto \tau$
0

Figure 24. Controller for the CKBot problem (plugs matrix)

So far, the only ways to specify a DD were: a linear order (that does not exist for Figure 23), a DD *example* (for DDDs and SDDs), or the full definition of a new category (ΣDDs). polyDD provides a formal and graphical specification suitable for complex DD descriptions.

VI. Conclusion

This article introduces polyDD, a category of DDs generalizing a wide range of existing ones. By defining general principles and through examples, we show polyDD covers the features of most existing DD categories (*i.e.* vanishing terminals, transparent domains, and the sharing of representations).

By lack of space, we focus our explanations on the data structure itself and only provide hints concerning the associated operations. We deliberately use an informal style, concentrating our explanation on the power of the polyDDs and their principles.

polyDD opens several important issues concerning: *i)* the required DD-engineering to evaluate for instance controllers efficiency (*i.e.* efficiency of a variable order) and *ii)* the generalization of optimizations developed for some DD categories.

This work is a first step towards a more complete generalization encompassing multi-terminals and multi-valued DD categories.

References

[1] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, “Symbolic Model Checking: 10^{20} States and Beyond,” in *5th Annual IEEE Symposium on Logic in Computer Science*, 1990, pp. 428–439.

[2] G. Jennings, J. Isaksson, and P. Lindgren, “Ordered Ternary Decision Diagrams and the Multivalued Compiled Simulation of Unmapped Logic,” in *27th IEEE Annual Simulation Symposium*, 1994, pp. 99–105.

[3] R. E. Bryant, “Graph-based algorithms for Boolean function manipulation,” *IEEE Trans. on Computers*, vol. 35, no. 8, pp. 677–691, 1986.

[4] S.-I. Minato, “Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems,” in *DAC ’93: 30th International Conference on Design Automation*, 1993, pp. 272–277.

[5] T. Kam, T. Villa, R. K. Brayton, and A. Sangiovanni-Vincentelli, “Multi-Valued Decision Diagrams: Theory and Applications,” *Multiple-Valued Logic*, vol. 4, no. 1–2, pp. 9–62, 1998.

[6] D. Buchs and S. Hostettler, “Σ Decision Diagrams,” in *TERMGRAPH 2009: 5th International Workshop on Computing with Terms and Graphs*, no. TR-09-05, 2009, pp. 18–32.

[7] M. Park, S. Chitta, A. Teichman, and M. Yim, “Automatic Configuration Recognition Methods in Modular Robots,” *International Journal of Robotics Research*, vol. 27, no. 3-4, pp. 403–421, 2008.

[8] R. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi, “Algebraic Decision Diagrams and Their Applications,” in *IEEE/ACM International Conference on CAD*, 1993, pp. 188–191.

[9] M. Wan and G. Ciardo, “Symbolic State-Space Generation of Asynchronous Systems Using Extensible Decision Diagrams,” in *SOFSEM ’09: 35th Conference on Current Trends in Theory and Practice of Computer Science*, 2009, pp. 582–594.

[10] K. Strehl and L. Thiele, “Symbolic Model Checking of Process Networks using Interval Diagram Techniques,” in *ICCAD ’98: International Conference on Computer-Aided Design*, 1998, pp. 686–692.

[11] J.-M. Couvreur, E. Encrenaz, E. Paviot-Adet, D. Poitrenaud, and P.-A. Wacrenier, “Data Decision Diagrams for Petri Net Analysis,” in *ATPN ’02: Applications and Theory of Petri Nets*, 2002, pp. 101–120.

[12] J.-M. Couvreur and Y. Thierry-Mieg, “Hierarchical Decision Diagrams to Exploit Model Structure,” in *FORTE ’05: 25th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems*, 2005, pp. 443–457.

[13] Y.-T. Lai and S. Sastry, “Edge-valued Binary Decision Diagrams for Multi-Level Hierarchical Verification,” in *DAC ’92: 29th International Conference on Design Automation*, 1992, pp. 608–613.

[14] G. Ciardo and R. Siminiceanu, “Using Edge-Valued Decision Diagrams for Symbolic Generation of Shortest Paths,” in *FMCAD ’02: Int. Conf. on Formal Methods in Computer-Aided Design*, 2002, pp. 256–273.

[15] J. Ossowski and C. Baier, “A Uniform Framework for Weighted Decision Diagrams and its Implementation,” *Int. Journal on Software Tools for Technology Transfer*, vol. 10, no. 5, pp. 425–441, 2008.

[16] M. Fujita, P. C. McGeer, and J. C.-Y. Yang, “Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation,” *Formal Methods in System Design*, vol. 10, no. 2-3, pp. 149–169.

[17] S. J. Friedman and K. J. Supowit, “Finding the Optimal Variable Ordering for Binary Decision Diagrams,” *IEEE Transactions on Computers*, vol. 39, no. 5, pp. 710–713, 1990.

[18] S. Tani, K. Hamaguchi, and S. Yajima, “The Complexity of the Optimal Variable Ordering Problem of Shared Binary Decision Diagrams,” in *ISAAC ’93: 4th International Symposium on Algorithms and Computation*, vol. 762, 1993.

[19] B. Bollig and I. Wegener, “Improving the Variable Ordering of OBDDs Is NP-Complete,” *IEEE Transactions on Computers*, vol. 45, no. 9, pp. 993–1002, 1996.