Multiway decision graphs for automated hardware verification

F. Corella[1] Z. Zhou[2] X. Song[3] M. Langevin[4]
E. Cerny[5]

August 10, 1994

[1]IBM T. J. Watson Research Center, corella@watson.ibm.com
[2]Dépt. d'I. R. O., Univ. de Montréal, zhouz@iro.umontreal.ca
[3]Dépt. d'I. R. O., Univ. de Montréal, song@iro.umontreal.ca
[4]GMD–SET, langevin@borneo.gmd.de
[5]Dépt. d'I. R. O., Univ. de Montréal, cerny@iro.umontreal.ca

Abstract

Traditional BDD-based methods of automated verification suffer from the drawback that they require a binary representation of the circuit. To overcome this limitation we propose a broader class of decision graphs, called *Multiway Decision Graphs* (MDGs), of which BDDs are a special case. With MDGs, a data value is represented by a single variable of abstract type, rather than by 32 or 64 boolean variables, and a data operation is represented by an uninterpreted function symbol. MDGs are thus much more compact than BDDs, and this greatly increases the range of circuits that can be verified. We give algorithms for MDG manipulation, and for implicit state enumeration using MDGs. We have implemented an MDG package and provide initial experimental results.

Keywords—Multiway Decision Graphs, Binary Decision Diagrams, Automated Hardware Verification, State Machine Verification, Reachability Analysis

1 Introduction

1.1 Limitations of BDD-based verification methods

Bryant's Reduced and Ordered Binary Decision Diagrams (BDDs) [3, 4] have proved to be a powerful tool for automated hardware verification [2, 5, 6, 9, 10, 16, 31]. BDDs, however, have a drawback: they require a binary representation of the circuit. Every individual bit of every data signal must be represented by a separate boolean variable, while the size of a BDD grows, sometimes exponentially, with the number of variables. This means that BDD-based verification methods often take too long, or run out of memory, when applied to circuits having a complex data path. As an example of this difficulty, BDDs cannot be used to compare two circuits containing multipliers [4], even if the two circuits use *identical* multipliers.

To overcome this limitation we propose a broader class of decision graphs, called *Multiway Decision Graphs* (MDGs), of which BDDs are a special case. With MDGs, a data value is represented by a single variable of abstract type, rather than by 32 or 64 boolean variables, and a data operation is represented by an uninterpreted function symbol. MDGs are thus much more compact than BDDs for circuits having a data path, and this greatly increases the range of circuits that can be verified.

1.2 Related work

Interactive verification by theorem proving [8, 20, 22] does not require a boolean representation of the circuit: it is usually carried out at a higher level of abstraction. Indeed part of the inspiration for our work comes from prior work on interactive verification, and in particular from the fact that Joyce [22] verified the Tamarack-3 microprocessor at such a high-level of abstraction that he did not even mention the width of the data path. This is in striking contrast with BDD-based methods, where an increase in the width of the data path often makes verification impossible. But interactive verification has its own drawback, viz. the user is responsible for coming up with the proof of correctness and feeding it to the theorem prover, which can be quite difficult and time consuming. Our work attempts to raise the level of abstraction of automated verification methods to that of interactive methods, without sacrificing automation. (Naturally, we have used Joyce's

Tamarack for our initial case studies.)

In the automated verification community the difficulties faced by boolean methods when verifying circuits with a substantial data path are well known, and have been tackled by many researchers.

Clarke, Grumberg, and Long [11, 28] have shown examples where verifications problems involving circuits with wide data paths can be reduced by a *data abstraction* technique to simpler problems involving *parameterized circuit descriptions* where the data path is only a few bits wide. These simpler problems can then be solved by BDD-based model checking. However, the fact that correctness of the simpler circuit implies correctness of the original circuit is not always obvious, and is not verified mechanically. Also, the data abstraction function has to be provided by the user; this may require considerable ingenuity, and has to be done anew for each verification problem. Similarly, Kurshan [23] has proposed the use of *homomorphic reductions* to simplify verification problems stated as tests of ω-language containment. Again, each problem requires its own homomorphism, which has to be provided by the user. In contrast to these two *problem reduction* techniques, MDGs afford new *problem resolution* techniques, which expand the range of circuits that can be verified directly, without recourse to ingenious problem transformations.

Wolper [32] has shown that *data independent systems* can be verified by reducing the domain of data values to a very small set. But data independent systems are essentially systems that transfer data without observing it or performing any computation on it, and thus the method is not widely applicable.

In some cases it is possible to restate a verification problem concerning a circuit that consists of a data path and a controller in terms of the controller only [1, 17]. In this case the CPU time needed for verification is of course independent of the width of the data path, which is not the case for the data abstraction method of [11]. However this is practically feasible only in special cases, and the equivalence of the original problem to the restated one is not verified mechanically.

MDGs have their roots in the work of Langevin and Cerny [24, 25, 26, 27] and Corella [13, 14], who have concurrently and independently developed similar techniques based on the use of variables of abstract type to denote data values and uninterpreted function symbols to denote data operations. More recently, Burch and Dill [7] have also proposed a verification method that uses uninterpreted function symbols to denote data operations. These techniques can deal with data path feedback and can verify the control circuitry and the data path together, without getting bogged down in the details of data path operations, in time independent of the width of the data path. They are applicable when data path operations can be viewed as black boxes, i.e., when the specification and the implementation use the same data path operators, and correctness does not depend on the meaning of those operators. Verification is then carried out at a high level in the design hierarchy, without taking into account the implementations of the data path components. The approach is well-suited for verifying general purpose microprocessors, as well as circuits produced by high-level synthesis, since in both cases data operations are indeed viewed as black boxes.

However, while these methods are better than BDD methods at handling the data path, they seem less good than BDD methods when it comes to sequential verification of circuits with complex control. BDDs can compactly represent sets of states; this is the basis for the technique of *implicit state enumeration*, which has been used very successfully for verification of I/O equivalence, and model checking [2, 5, 6, 9, 16, 31]. In contrast, the method of [13, 14] reduces to *explicit* enumeration when there is no data path, and the method of [7] does not seem to apply to problems requiring state space traversal. Langevin and Cerny use *Extended Ordered BDDs* (EOBDDs) [27], which could conceivably be used for implicit enumeration, but the state traversal method of [25, 26] is similar to that of [13, 14], and also reduces to explicit enumeration in the absence of a data path. (EOBDDs are the immediate precursors of MDGs. The difference between EOBDDs and MDGs is discussed below in Section 3.)

On the other hand MDGs combine the advantages afforded by abstract types and uninterpreted function symbols for representing the data path with the advantages of decision graphs for dealing with complex control, including the possibility of implicit enumeration of the states of a sequential circuit. This makes it possible to lift many BDD techniques from the boolean domain to higher levels of abstraction. While the methods of [14, 25] could be called *abstract explicit enumeration* methods,[1] we describe in Section 6.2.1 a state exploration method that could be called *abstract implicit enumeration*.

[1]Or, better, *explicit symbolic enumeration*; but this would be confusing because the word *symbolic* has been misused to refer to BDD-based methods.

MDGs are similar in name and structure to the *Multivalued Decision Diagrams* (MDDs) of [30], but the similarity is superficial. MDDs and MDGs have in common that any number of edges can issue from a given node. In MDDs, however, those edges are labeled by constants that denote pairwise distinct values comprising the entire range of values for the node. In MDGs the labels of the edges can be first-order terms, need not be mutually exclusive, and need not denote all the values in a given range. This makes it possible to use variables of abstract type and uninterpreted function symbols in MDGs, which is not possible in MDDs.[2]

1.3 Outline

While BDDs are graphical representations of formulas of propositional logic, MDGs represent formulas of the predicate calculus. In Section 2 we briefly describe a version of first-order logic that features a distinction between abstract and concrete types, which makes it well-suited for giving MDGs their meaning. Then in sections 3 and 4 we motivate the specific structure of MDGs and provide the formal definition. In Section 5 we describe a basic set of algorithms for MDG manipulation. In Section 6 we give an algorithm for state space traversal based on implicit state enumeration with MDGs, and we discuss the use of MDGs for combinational and sequential verification. Section 7 provides experimental results obtained with an MDG package implemented in Prolog, and Section 8 summarizes the contributions of the paper. The appendix contains detailed descriptions of the MDG algorithms.

2 Formal logic

2.1 Syntax

As in ordinary many-sorted first-order logic, the vocabulary consists of *sorts*, *constants*, *variables*, and *function symbols* (or *operators*). Constants and

[2]A related difference is that MDDs represent terms (functions), while MDGs represent sentences (relations): the functional character of MDDs requires that one and only one value be specified by the graph for each combination of node values. As for our choice of name, since MDGs do not denote functions they are *multiway* rather than *multivalued*; and we call them *graphs* rather than *diagrams* to obtain a different acronym.

variables have sorts. An n-ary function symbol ($n > 0$) has a type $\alpha_1 \times \ldots \times \alpha_n \to \alpha_{n+1}$, where $\alpha_1 \ldots \alpha_{n+1}$ are sorts. The (well-typed) terms and their sorts are defined as usual. The *top symbol* of a term is defined as follows: the top symbol of $f(A_1, \ldots, A_n)$ is f, and the top symbol of a term consisting of a single occurrence of a variable or a constant is that variable or constant.

We deviate from standard many-sorted first-order logic by introducing a distinction between *concrete* (or *enumerated*) sorts, and *abstract sorts*; the difference is that concrete sorts have *enumerations*, while abstract sorts do not. The enumeration of a concrete sort α is a set of distinct constants of sort α. We refer to constants occurring in enumerations as *individual constants*, and to other constants as *generic constants*. We shall view generic constants as 0-ary function symbols. Individual constants are overloaded, they can appear in more than one enumeration. If X is a set (or "vector") of variables, we write X_{conc} and X_{abs} to denote the sets of elements of X that are variables of concrete and abstract sort respectively.

The distinction between abstract and concrete sorts leads to a distinction between three kinds of function symbols. Let f be a function symbol of type $\alpha_1 \times \ldots \times \alpha_n \to \alpha_{n+1}$. If α_{n+1} is an abstract sort then f is an *abstract function symbol.* If all the $\alpha_1 \ldots \alpha_{n+1}$ are concrete, f is a *concrete* function symbol. If α_{n+1} is concrete while at least one of $\alpha_1 \ldots \alpha_n$ is abstract, then we refer to f as a *cross-operator*; cross-operators, as we shall see, are useful for modeling feedback from the data path to the control circuitry. Concrete function symbols, which are used in [13, 14], must have explicit definitions; they can therefore be eliminated and do not appear in MDGs. Abstract function symbols and cross-operators are *uninterpreted.* Partial information about them could be provided in the form of (conditional) equations to be used as (conditional) *rewrite rules.* We hope to be able to provide rewriting algorithms for MDGs in the near future.

A term that has no concrete subterms other than individual constants is said to be *concretely-reduced.* A term of the form "$f(A_1, \ldots, A_n)$" where f is a cross-operator and $A_1 \ldots A_n$ are concretely-reduced terms, is a *cross-term.* As the special case $n = 0$, a term "c" consisting of a single generic constant c of concrete sort is also a cross-term. Note that no concrete variable can occur in a concretely-reduced term or in a cross-term.

An *equation* is an expression "$A_1 = A_2$" where A_1 and A_2 are terms of same type α. *Atomic formulas* are the equations, plus $\mathbf{T}$ (truth) and $\mathbf{F}$ (falsity). *Formulas* are built from the atomic formulas in the usual way using logical connectives and quantifiers.

2.2 Semantics

An *interpretation* is a mapping ψ that assigns a denotation to each sort, constant and function symbol and satisfies the following conditions:

1. The denotation $\psi(\alpha)$ of a sort α is a non-empty set.

2. If c is a constant—individual or generic—of sort α, then $\psi(c) \in \psi(\alpha)$.

3. If f is a function symbol of type $\alpha_1 \times \ldots \times \alpha_n \rightarrow \alpha_{n+1}$, then $\psi(f)$ is a function from the cartesian product $\psi(\alpha_1) \times \ldots \times \psi(\alpha_n)$ into the set $\psi(\alpha_{n+1})$.

4. If a and a' are distinct individual constants in the enumeration of a concrete sort α, then $\psi(a)$ and $\psi(a')$ are *distinct* elements of the set $\psi(\alpha)$.

5. If $a_1 \ldots a_n$ are the individual constants in the enumeration of a concrete sort α, then $\psi(\alpha)$ coincides with the set $\{\psi(a_1), \ldots, \psi(a_n)\}$.

X being a set of variables, a *variable assignment* with domain X compatible with an interpretation ψ is a function ϕ that maps every variable $x \in X$ of sort α to an element $\phi(x)$ of $\psi(\alpha)$. The denotation of a term and the truth or falsity of a formula under an interpretation and a compatible variable assignment are defined as usual. We write $\psi, \phi \models P$ if a formula P denotes truth under an interpretation ψ and a ψ-compatible variable assignment ϕ to the variables that occur free in P. We write $\psi \models P$ if $\psi, \phi \models P$ for all such assignments ϕ, and $\models P$ if $\psi, \phi \models P$ for all appropriate ψ and ϕ. Two formulas P and Q are *logically equivalent* iff

$$\models P \Leftrightarrow Q. \tag{1}$$

2.3 Herbrand models

Let X be a set of variables of abstract sort, containing at least one variable of each abstract sort. An interpretation ψ is said to be a *Herbrand interpretation* generated by X if it satisfies the following conditions:

1. The denotation of a concrete sort α is the set of individual constants in the enumeration of α.

2. The denotation of an abstract sort α is the set of concretely-reduced terms of sort α with variables in X.

3. The denotation of an abstract function symbol f of type $\alpha_1 \times \ldots \times \alpha_n \rightarrow \alpha_{n+1}$ is the function that maps every n-tuple of terms $A_1 \ldots A_n$ of sorts $\alpha_1 \ldots \alpha_n$ with variables in X to the term $f(A_1, \ldots, A_n)$. The denotation of a generic constant c of abstract sort is the constant c itself (more precisely, the term consisting of a single occurrence of c).

4. The denotation of a cross-operator f of type $\alpha_1 \times \ldots \times \alpha_n \rightarrow \alpha_{n+1}$ is an *arbitrary* function that maps every n-tuple of terms $A_1 \ldots A_n$ of sorts $\alpha_1 \ldots \alpha_n$ with variables in X to an individual constant in the enumeration of the concrete sort α_{n+1}. The denotation of a generic constant c of concrete sort α is an *arbitrary* individual constant in the enumeration of α (more precisely, the term consisting of a single occurrence of that constant).

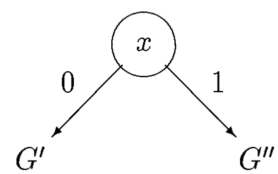
Note that the union of the denotations of the sorts, called the *Herbrand universe*, is the set of concretely-reduced terms with variables in X.

Let X be a set of variables, this time of abstract or concrete sort, containing at least one variable of each abstract sort; and let Y be a set of variables disjoint from X. We say that $M = (\psi, \phi)$ is a *Herbrand model* of type $X \hookrightarrow Y$ iff ψ is a Herbrand interpretation generated by X_{abs} and ϕ is a variable assignment compatible with ψ, with domain $X \cup Y$, that maps every $x \in X_{\mathsf{abs}}$ to itself. Note that the Herbrand universe for ψ is the set of concretely-reduced terms with variables in X_{abs}, and that each such term denotes itself under ψ and ϕ. If $f(A_1, \ldots, A_n)$ is a cross-term with variables in X_{abs}, the arguments $A_1 \ldots A_n$ are elements of the Herbrand universe that denote themselves under ψ and ϕ; hence the denotation of the cross-term itself (an individual constant) is simply $\psi(f)(A_1, \ldots, A_n)$. The following proposition is an immediate consequence of this:

Proposition 1 Let X be a set of variables containing at least one variable of each abstract sort, and Y a set of variables disjoint from X. A Herbrand model of type $X \hookrightarrow Y$ is uniquely determined by the denotations of (i) the cross-terms with variables in X, (ii) the abstract variables in Y, and (iii) the concrete variables in $X \cup Y$.

3 From BDDs to MDGs

Consider the BDD G:



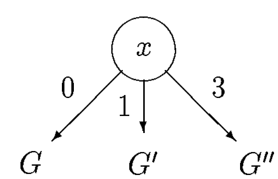
where G' and G'' are subgraphs of G. Traditionally, if G' and G'' represent the formulas P' and P'' respectively, then G is viewed as representing the formula P:

$$((\neg x) \wedge P') \vee (x \wedge P''). \tag{2}$$

However, it can also be viewed as representing the equivalent formula

$$(x = 0 \wedge P') \vee (x = 1 \wedge P''). \tag{3}$$

This suggests a generalization of the notion of decision graph: there is no need for x to range over the set $\{0, 1\}$. Furthermore, there is no need for the labels of the edges to exhaustively denote all the possible values of x. For example, x could range over $\{0, 1, 2, 3\}$, and there could be, say, only three edges issuing from the root, as in the following graph:

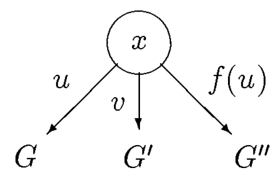


If G, G', and G'' represent the formulas P, P', and P'' respectively, then this graph could represent the formula

$$(x = 0 \wedge P) \vee (x = 1 \wedge P') \vee (x = 3 \wedge P''). \tag{4}$$

When x denotes 2, this formula is simply a false sentence. Finally, there is no need for the edges to be mutually exclusive.

It is then possible to let nodes range over abstract sorts for which there is no exhaustive set of edges, and to use non-mutually-exclusive first-order terms as edge labels. For example, if x, u, and v are variables of abstract type α, f is a function symbol of type $\alpha \to \alpha$, and G, G', and G'' represent P, P', and P'' respectively, then the graph



represents the formula

$$(x = u \wedge P) \vee (x = v \wedge P') \vee (x = f(u) \wedge P''). \tag{5}$$

This freedom in labeling the edges distinguishes MDGs from other kinds of decision graphs. We have already seen that it distinguishes MDGs from MDDs. Langevin and Cerny's EOBDDs [27] allow for boolean variables and variables of abstract type. But each internal node must have two edges labeled 0 and 1. Hence internal nodes can only be labeled by boolean variables or boolean conditions. Thus boolean variables and abstract variables play asymmetric roles in EOBDDs. In contrast, internal MDG nodes can be labeled by variables of any sort.

4 Definition and canonicity

The observations of Section 3 lead to the following preliminary definition:

Definition 1 A *multiway decision graph* (MDG) is a finite directed acyclic graph (DAG) G where the leaf nodes are labeled by formulas, the internal nodes are labeled by terms, and the edges issuing from an internal node N are labeled by terms of same sort as the label of N. Such a graph represents a formula defined inductively as follows: (i) if G consists of a single leaf node labeled by a formula P, then G represents P; (ii) if G has a root node labeled A with edges labeled $B_1 \ldots B_n$ leading to subgraphs $G'_1 \ldots G'_n$, and if each G'_i represents a formula P_i, then G represents the formula $\bigvee_{1 \leq i \leq n}((A = B_i) \wedge P_i)$.

But this definition is of course too general. We have to provide a carefully chosen set of *well-formedness conditions* to turn MDGs into *canonical representations* that can be manipulated by *efficient algorithms*.

4.1 Well-formedness conditions

Bryant's well-formedness conditions on BDDs (viz. the requirements that they be *reduced* and *ordered*) and our prior experience with verification using abstract types and EOBDDs suggest the following well-formedness conditions for MDGs:

1. *Kinds of nodes.* An internal node must be labeled by a variable of abstract sort, with edges issuing from the node labeled by concretely-reduced terms of that same sort; or by a variable of concrete sort, with edges labeled by individual constants in the enumeration of that sort; or by a cross-term, with edges labeled by individual constants in the enumeration of the sort of the cross-term. A leaf node must be labeled by **T**, except in the case where the graph has only one node labeled **F**.

We refer to an occurrence of a variable in a term that labels an edge or in a cross-term that labels a node as a *secondary occurrence*, while an occurrence of a variable as the label of a node is a *primary occurrence*. Since the edge labels and the subterms of a cross-term are concretely-reduced terms, they can contain no concrete variables, and thus only abstract variables can have secondary occurrences. The *primary variables* (resp. *secondary variables*) of a graph G are those that have primary (resp. secondary) occurrences in G.

2. *Ordering.* The labels of the edges issuing from a given node must appear in a *standard term order*, without repetitions. Along each path, the variables and the cross-operators of the cross-terms that label the nodes must appear in a *custom symbol order*, and cross-terms with same cross-operator must appear in the standard term order; there must be no repeated labels.

The custom symbol order is a generalization (to include cross-operators) of the variable ordering used for BDDs, and plays the same role. It need only involve the cross-operators and those variables that may appear as node labels. It is chosen carefully for each particular application so as to keep the

MDGs of manageable size if possible. The standard term ordering, on the other hand, is chosen arbitrarily once and for all;[3] it need not be compatible with the custom symbol order. From these two orderings we define the *node-label ordering* among the variables and cross-terms as follows: A comes before B iff the top symbol of A comes before the top symbol of B, or A and B are cross-terms with same cross-operator, and A comes before B in standard term order. Condition 2 states that node labels must appear in node-label order along each path.

3. *Minimality.* There must be no distinct isomorphic subgraphs, and no *redundant nodes.*

In an MDG, a *redundant node* is a node labeled by a concrete variable or a cross-term of sort α, with edges labeled by all the individual constants in the enumeration of α and leading to the same subgraph.

These conditions are not enough, however. To obtain a canonical representation we add the following one:

4. No variable should have both primary and secondary occurrences in the same graph.[4]

Finally, conditions 5 and 6 are motivated by the need to compute disjunction and conjunction respectively, as we shall see below in Section 5.

5. The set of abstract variables having primary occurrences along a path is the same for all paths in a given graph.

6. If a node N is labeled by an abstract variable y, and an abstract variable x that participates in the custom symbol order occurs in a term A that labels one of the edges that issue form N, then x must come before y in the custom symbol order. Similarly, if N is labeled by a cross-term A with cross-operator f, and x is an abstract variable that occurs in A, then x must come before f in the custom symbol order.

[3]This does not mean that the standard term ordering has no influence on BDD size: it does have an influence. For the time being we consider using only the symbol order as a means of controlling BDD size, but this may change in the future.

[4]To obtain a canonical representation it is sufficient to rule out variables having primary and secondary occurrences along the same path. But this together with condition 5 below implies the stated condition.

This completes the set of well-formedness conditions on MDGs.

Definition 2 An MDG G is said to be *well-formed* iff it satisfies conditions 1 through 6 above.

Observe that BDDs are a special case of MDGs. More precisely, a BDD can be turned into an MDG by changing the leaves labeled 1 to leaves labeled **T**, and removing the leaves labeled 0 and the edges leading to them; except, of course, for the BDD 0, which corresponds to the MDG **F**.

Let X and Y be disjoint sets of variables. We shall say that an MDG G is of type $X \rightarrow Y$ (resp. $X \rightsquigarrow Y$) iff (i) the set of abstract primary variables of G *is equal to* (resp. *is a subset of*) Y_{abs}, (ii) the set of secondary variables of G is a subset of X_{abs}, and (iii) the set of concrete variables having occurrences in G is a subset of $X_{\mathrm{conc}} \cup Y_{\mathrm{conc}}$. Whenever we use either notation, it will be understood that X and Y are disjoint.[5]

From now on *MDG* will mean *well-formed MDG* unless otherwise stated. We shall refer to the graphs consisting of a single leaf node labeled **F** or **T** simply as the graphs **F** and **T**.

4.2 MDGs as canonical representations

We have been using the word *path* to refer, in and MDG, to a sequence of consecutive nodes and edges starting at a the root of the graph and ending at a leaf node. A *path prefix* is a sequence of consecutive nodes and edges that starts at a the root and ends at a node that may be an internal node. A *linear MDG* is an MDG that has a single path. Two linear MDGs L and L' *diverge* if they have isomorphic path prefixes π and π' that lead to internal nodes N and N' where N and N' have the same label, but the edge issuing from N in L has a different label than the edge issuing from N' in L'.

Lemma 1 Every MDG other than **F** is satisfiable. More precisely, if G is an MDG of type $X \rightsquigarrow Y$ other than **F**, with X containing at least one variable of each abstract type, then there exists a Herbrand model of type $X \hookrightarrow Y$ that satisfies G.

[5]According to this definition, **T** and **F** are graphs of type $X \rightsquigarrow Y$ for every X and Y, but not of type $X \rightarrow Y$.

PROOF. Note that if A is the label of an edge of G, or an immediate subterm of a cross-term that labels a node of G, then A is an element of the Herbrand universe that denotes itself in any Herbrand model of type $X \hookrightarrow Y$.

Let π be a path of G. By Proposition 1, there exists a Herbrand model M of type $X \hookrightarrow Y$ where the denotation of each variable or cross-term A that labels a node of π is the label of the corresponding edge. (Variables in $Y \cup X_{\mathrm{conc}}$ and cross-terms with variables in X that do not appear in π can have arbitrary denotations.) M satisfies π and hence G.
$\Box$

Lemma 2 Let L and L' be divergent linear MDGs of type $X \rightsquigarrow Y$, and let M be a Herbrand model of type $X \hookrightarrow Y$. Then if M satisfies L, it falsifies L'.

PROOF. Let N and N' be the nodes of π and π' where they diverge, let A be the label of N, and let B, B' be the edges that issue from N in π and π' respectively. Since M satisfies π it satisfies the equation $A = B$. But then, since B and B' denote themselves and are distinct, it falsifies the equation $A = B'$. Hence it falsifies π'.
$\Box$

Theorem 1 Let G and G' be two MDGs representing formulas P and P', and such that no abstract primary variable of G has a secondary occurrence in G' or *vice versa*. Then, if P and P' are logically equivalent, G and G' are isomorphic.

PROOF. First, by Lemma 1, if P and P' are logically equivalent and either is $\mathbf{F}$, the other is also $\mathbf{F}$, and G and G' are isomorphic.

Now assume that G and G' are as stated, and neither of them consists of a single node labeled $\mathbf{F}$. Let Y be the set of variables having primary occurrences in G or G', and let X be a set of abstract variables, disjoint from Y, containing all variables having secondary occurrences in either graph, plus, for each abstract sort α, a variable v_α of sort α that does not occur in G or G'. G and G' are then MDGs of type $X \rightsquigarrow Y$. Assume that G and G' are not isomorphic, and let us prove that P and P' are not equivalent.

The reader will easily verify that, by adding redundant nodes as needed, it is possible to construct "completions" G_c and G'_c of G and G' satisfying the following conditions: they are well-formed MDGs of type $X \rightsquigarrow Y$ except for the fact that they may have redundant nodes; they represent formulas P_c and P'_c that are logically equivalent to P and P' respectively; the set of concrete node labels is the same along all the paths of both graphs; and, because they are obtained by adding redundant nodes to G and G', which have no redundant nodes themselves and are not isomorphic, they are not isomorphic.

By eliminating all structure sharing, it is possible to derive from G_c and G'_c trees T_c and T'_c representing the same formulas P_c and P'_c; T_c and T'_c are well-formed MDGs of type $X \rightsquigarrow Y$, except that they may not satisfy condition 3. We leave it to the reader to show that, because G_c and G'_c are not isomorphic and each of them has no distinct isomorphic subgraphs, T_c and T'_c are not isomorphic. (Hint: a tree can be transformed into a homomorphic DAG having no distinct isomorphic subgraphs by a structural reduction algorithm that produces a reduction table (usually implemented as a *hash table* in a practical algorithm); if two trees are isomorphic they yield the same table, which can be used do define an isomorphism between the DAGs.)

Since T_c and T'_c are not isomorphic, there must be isomorphic path prefixes π and π' (possibly of zero length) in T_c and T'_c leading to nodes N and N' such that:

1. One of the nodes, say N, is an internal node labeled A while the other, N', is a leaf node labeled $\mathbf{T}$ or an internal node labeled A', with A' coming after A in node-label order. Or,

2. N and N' are internal nodes having the same label A, but there is an edge issuing from one of the nodes, say N', labeled by a term B that does not appear among the labels of the edges that issue from the other node, N.

In both cases we are going to show that P_c and P'_c are not equivalent by constructing a Herbrand model that satisfies P'_c but falsifies P_c.

Consider case 1. Since all paths in T_c and T'_c have the same set of concrete node labels and A does not appear as a node label in the path prefix π', A must be a variable $y \in Y$ of abstract sort α.

Let π'' be a path in T'_c having π' as a prefix. We saw in the proof of Lemma 1 how to construct a Herbrand model M of type $X \hookrightarrow Y$ that satisfies

π'' and hence P'_c. Since y does not appear as a node label in π', we can choose its denotation arbitrarily: let it be v_α. By the argument used in the proof of Lemma 2, M falsifies every path in T_c that diverges from π, and, since v_α is not a label of an edge issuing from N in T_c, it also falsifies every path having π as a prefix. Thus M falsifies P_c.

Now consider case 2. Let again π'' be a path of T'_c having π' as a prefix. By Lemma 1, there exists a Herbrand model M of type $X \hookrightarrow Y$ that satisfies π'' and hence P'_c. But π'' diverges from every path in P_c. Therefore, by Lemma 2, M falsifies every path in T'_c, and hence falsifies P'_c.
□

5 MDG algorithms

BDD operations can be implemented using a single generic algorithm *Apply* [4]. This is because the two edges that issue from a BDD node labeled x span the range of values $\{0, 1\}$ that x can take, and this makes it possible to reason by cases. Since MDGs do not enjoy this property, a separate algorithm must be provided for each operation. We informally sketch below the algorithms for disjunction and relational product, and an algorithm called *pruning-by-subsumption* that provides a termination test for state space exploration in the presence of abstract types. Our goal in this section is only to provide an intuitive understanding of the algorithms. More detailed descriptions together with a proof of soundness of the pruning algorithm can be found in the appendix.

5.1 Disjunction

Consider two MDGs with the same root label, such as, for example,

where B_2 and B_3 are the only labels shared by the root edges. Let $P_1 \ldots P_6$ be the formulas represented by $G_1 \ldots G_6$. Then the depicted MDGs represent the formulas

$$(A = B_1 \wedge P_1) \vee (A = B_2 \wedge P_2) \vee (A = B_3 \wedge P_3) \tag{6}$$

and

$$(A = B_2 \wedge P_4) \vee (A = B_3 \wedge P_5) \vee (A = B_4 \wedge P_6). \tag{7}$$

If

$$\models P' \Leftrightarrow (P_2 \vee P_4) \tag{8}$$

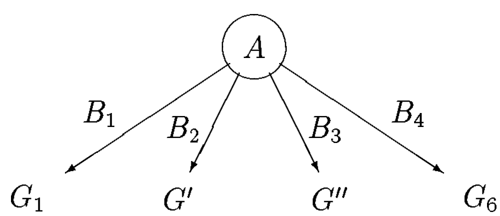
and

$$\models P'' \Leftrightarrow (P_3 \vee P_5) \tag{9}$$

then

$$\begin{aligned} \models \;& (((A = B_1 \wedge P_1) \vee (A = B_2 \wedge P_2) \vee (A = B_3 \wedge P_3)) \vee \\ & ((A = B_2 \wedge P_4) \vee (A = B_3 \wedge P_5) \vee (A = B_4 \wedge P_6))) \Leftrightarrow \\ & ((A = B_1 \wedge P_1) \vee (A = B_2 \wedge P') \vee (A = B_3 \wedge P'') \vee (A = B_4 \wedge P_6)). \end{aligned} \tag{10}$$

Thus if we can recursively compute MDGs G' and G'' representing the formulas P' and P'', the disjunction of the graphs shown above will be represented by the following graph:



On the other hand, if two MDGs have different root labels, it may not be possible to compute their disjunction. For example, it can be shown that, if x and y are distinct abstract variables, and a and b are distinct generic constants of abstract sort, there exists no well-formed MDG representing a formula that is logically equivalent to $(x = a) \vee (y = b)$. Hence it is not possible to compute the disjunction of the following MDGs:[6]

[6] We shall see that, in some applications, it is useful to consider that the secondary variables of some MDGs are implicitly existentially quantified. Such an MDG can always be "completed" by adding nodes labeled by "missing" primary abstract variables, each with an edge labeled by a fresh secondary variable, implicitly existentially quantified. For

However, if two graphs have different root labels but the label that comes first in node-label order is a *concrete* variable x, we can revert to the case where the labels are the same by adding a redundant node labeled x at the top of the graph not labeled by x. We thus know how to compute the disjunction of two MDGs G and G' if G and G' have the same set of abstract variables along all their paths. Indeed, the roots of G and G' must then have the same label, or else the label that comes first in node-label ordering must be a concrete variable; and recursive computations of disjunction will apply to subgraphs of G and G' for which this is also the case. A detailed description of a disjunction algorithm based on this method (in an n-ary version) can be found in Section A.2.

The feasibility of computing disjunction is the motivation for the fifth well-formedness condition: if an MDG does not have the same set of abstract variables along all its paths it will not be possible to compute its disjunction with any MDG other than **F** or **T** by the method that we have outlined.

5.2 Conjunction

Consider again two MDGs

example, the first MDG in the figure can be completed by adding a node labeled y with an edge labeled by a fresh variable u. If u is considered to be existentially quantified, then the completed MDG represents the formula $(\exists u)(x = a \wedge y = u \wedge \mathbf{T})$, which is logically equivalent to the formula $(\exists u)(x = a \wedge \mathbf{T})$ represented by the original MDG. Similarly, the second MDG can be completed by adding a node labeled x with an edge labeled by a fresh abstract variable v. Once the two MDGs have the same set of primary abstract variables, there is no difficulty in computing their disjunction.

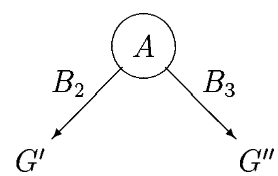
where, as before, $G_1 \ldots G_6$ represent $P_1 \ldots P_6$ and B_i and B_j are distinct terms for $1 \leq i < j \leq 4$. If the common root label A is of *concrete* sort α (either a concrete variable or a cross-term), then B_1, B_2, B_3, and B_4 are individual constants in the enumeration of α, and $\psi(B_i) \neq \psi(B_j)$ for $i \neq j$. Hence

$$\models \neg(B_i = B_j) \tag{11}$$

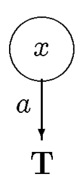
for $i \neq j$, and

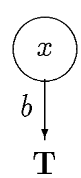
$$\begin{aligned} \models \;\; & (((A = B_1 \wedge P_1) \vee (A = B_2 \wedge P_2) \vee (A = B_3 \wedge P_3))\wedge \\ & ((A = B_2 \wedge P_4) \vee (A = B_3 \wedge P_5) \vee (A = B_4 \wedge P_6))) \Leftrightarrow \\ & ((A = B_2 \wedge P_2 \wedge P_4) \vee (A = B_3 \wedge P_3 \wedge P_5)) \end{aligned} \tag{12}$$

Thus if we can recursively compute two MDGs G' and G'' representing $P_2 \wedge P_4$ and $P_3 \wedge P_5$, the conjunction of the depicted MDGs is represented by the MDG:



On the other hand, it may be impossible to compute the conjunction of two MDGs whose root nodes have the same label when that label is an *abstract variable*. For example, it is easy to see that there is no MDG representing a formula that is logically equivalent to $(x = a) \wedge (x = b)$ if x is an abstract variable and a and b are distinct generic constants of same abstract sort as x. Hence it is not possible to compute the conjunction of the following MDGs:





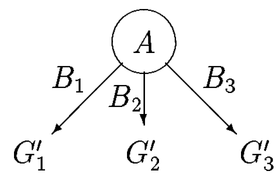
Now consider the case of two graphs G_1 and G_2 where the root label of G_1 comes before the root label of G_2 in the node-label ordering. Suppose, for example, that G_1 represents the formula

$$(A = B_1 \wedge P_1) \vee (A = B_2 \wedge P_2) \vee (A = B_3 \wedge P_3) \tag{13}$$

and G_2 represents a formula Q. We have:

$$\models \quad (((A = B_1 \wedge P_1) \vee (A = B_2 \wedge P_2) \vee (A = B_3 \wedge P_3)) \wedge Q) \Leftrightarrow \\ ((A = B_1 \wedge P_1 \wedge Q) \vee (A = B_2 \wedge P_2 \wedge Q) \vee (A = B_3 \wedge P_3 \wedge Q)) \tag{14}$$

Hence if G'_1, G'_2, and G'_3 represent the formulas $P_1 \wedge Q$, $P_2 \wedge Q$, and $P_3 \wedge Q$ respectively, the graph G_3:



represents a formula that is logically equivalent to the conjunction of the formulas represented by G_1 and G_2.

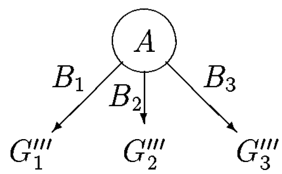
However, in the case where the label A of G_1 is an abstract variable x, this graph may not be well-formed. This is because x may have secondary occurrences in G_2, and hence it may also have secondary occurrences in G'_1, G'_2, or G'_3; therefore x may have both primary and secondary occurrences in G_3, contradicting the fourth well-formedness condition.

Symmetrically, if a variable y with primary occurrences in G_2 had occurrences in the edge labels B_1, B_2, or B_3, it could end up having both primary and secondary occurrences in G_3, again contradicting condition 4. But this possibility is ruled out: if y has a primary occurrence in G_2 then, given that the root label of G_2 comes after x in the node-label order, y must come after x in the node-label order, and hence in the custom symbol order; then by well-formedness condition 6 it cannot have occurrences in the edge labels B_1, B_2, B_3. *Ruling out this possibility is the motivation for well-formedness condition 6.*

To obtain a well-formed graph representing the conjunction of G_1 and G_2, it thus suffices to eliminate x from G_2; this can be done by substituting B_i, $i = 1, 2, 3$, for x in G_2 before taking the conjunction of G_2 with the subgraph of G_1 that the edge labeled B_i leads to. Indeed, let Q'_1, Q'_2, and Q'_3 be the formulas obtained by replacing x in Q with B_1, B_2, and B_3 respectively. Then by the rule of context-sensitive substitution [12] we have

$$\models \quad ((x = B_1 \wedge P_1 \wedge Q) \vee (x = B_2 \wedge P_2 \wedge Q) \vee (x = B_3 \wedge P_3 \wedge Q)) \Leftrightarrow \\ ((x = B_1 \wedge P_1 \wedge Q'_1) \vee (x = B_2 \wedge P_2 \wedge Q'_2) \vee (x = B_3 \wedge P_3 \wedge Q'_3)). \tag{15}$$

Assume that we can compute MDGs G_1'', G_2'', and G_3'', representing the formulas Q_1', Q_2', and Q_3' resulting from the substitutions. Then if we can recursively compute MDGs G_1''', G_2''', and G_3''', representing formulas logically equivalent to $P_1 \wedge Q_1'$, $P_2 \wedge Q_2'$, and $P_3 \wedge Q_3'$ respectively, the graph



is a well-formed MDG representing a formula logically equivalent to the conjunction of the formulas represented by G_1 and G_2.

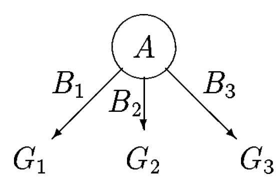
The computation of G_1'', G_2'', and G_3'' is not just a matter of replacing secondary occurrences of x in Q with the respective terms B_1, B_2, and B_3. Substitution in edge labels may require reordering the list of edges that issue from a given node, and removing duplicate edges. Subgraphs pointed to by edges whose labels become the same after substitution must be merged using the disjunction algorithm. Substitution in cross-terms may require node reordering. Because cross-terms with the same cross-operator label consecutive nodes, this reordering remains localized. Also, cross-terms that label distinct nodes may become the same after substitution. Along each path, this may result in redundant node-edge pairs (same node label, same edge label) and in contradictory node-edge pairs (nodes labeled by the same cross-terms, edges labeled by different individual constants). These redundancies and contradictions have to be eliminated.

Even without these complications, the computation of G_1'', G_2'', and G_3'' is an expensive operation because it requires traversal of G_2. To avoid this, we use a *lazy substitution* technique. Instead of performing the substitution immediately, we pass the information that x is to be replaced by B_i to the recursive computation of the conjunction of P_i with Q. Thus a list of pending substitutions is accumulated as the recursive computation progresses towards the leaves of the MDGs, and all these pending substitutions are applied at once to each abstract edge label and each cross-term. As an additional refinement, we filter the list of pending substitutions when each recursive call is made, removing all substitutions for variables that do not appear in the subgraphs to which the recursive call applies.[7]

We have thus outlined a method for computing the conjunction of two MDGs, applicable when the sets of primary variables of the two graphs are disjoint. It can be seen that the resulting MDG has primary variables that are among the primary variable of the conjuncts, including *all* abstract variables having primary occurrences in any of the conjuncts. The abstract variables having secondary occurrences in the result are among those having secondary occurrences in the conjuncts, excluding those having primary occurrences in any of the conjuncts. A detailed description of an algorithm based on this method can be found in Appendix A.3, which describes the relational product algorithm; conjunction is a special case of relational product.

5.3 Existential quantification

Existential quantification with respect to a variable x can be computed in the same way for an MDG as for a BDD when x is a primary variable of the graph. As an example, consider again an MDG with three root edges,



where G_1, G_2, G_3 represent three formulas P_1, P_2, P_3, and suppose we want to find an MDG representing the existential quantification of the formula represented by this graph with respect to a variable x that has primary occurrences in it.

By well-formedness condition 4, x cannot have secondary occurrences in the graph: it cannot occur in an edge label or in a cross-term labeling a node.

[7]In order to do this, we keep with each MDG node N the list of variables having secondary occurrences in the subgraph rooted at N. This overhead is justified because filtering the list of pending substitutions is essential. Indeed MDG algorithms take advantage of the structure sharing in MDGs through the use of result tables, as described in the appendix. The result table for conjunction (actually, for the more general relational product algorithm that we have implemented) must include the list of pending substitutions as an argument of the algorithm. This list must be as short as possible in order to have a good hit rate.

Thus either x is in fact the root label A, and then it does not occur in B_1, B_2, B_3, nor in P_1, P_2, P_3, or it is not, and then it does not occur in A, B_1, B_2, or B_3.

In the first case, since x does not occur in P_1, P_2, or P_3, the formula

$$(\exists x)((x = B_1 \wedge P_1) \vee (x = B_2 \wedge P_2) \vee (x = B_3 \wedge P_3)) \tag{16}$$

is logically equivalent to

$$((\exists x)(x = B_1) \wedge P_1) \vee ((\exists x)(x = B_2) \wedge P_2) \vee ((\exists x)(x = B_3) \wedge P_3). \tag{17}$$

But since x does not occur in B_i, $i = 1, 2, 3$, $(\exists x)(x = B_i)$ is logically equivalent to $\mathbf{T}$, and hence (16) is logically equivalent to

$$P_1 \vee P_2 \vee P_3. \tag{18}$$

In the second case the formula

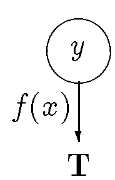
$$(\exists x)((A = B_1 \wedge P_1) \vee (A = B_2 \wedge P_2) \vee (A = B_3 \wedge P_3)) \tag{19}$$

is logically equivalent to

$$(A = B_1 \wedge (\exists x)P_1) \vee (A = B_2 \wedge (\exists x)P_2) \vee (A = B_3 \wedge (\exists x)P_3). \tag{20}$$

This provides a recursive procedure for computing existential quantification of an MDG with respect to one of its primary variables.

On the other hand, no algorithm is provided for existentially quantifying secondary variables. It is often not possible to compute existential quantification with respect to a variable with secondary occurrences. For example, the graph



represents the formula $y = f(x) \wedge \mathbf{T}$, and there is no MDG representing the formula $(\exists x)(y = f(x))$. In practice, however, this is not a problem because existential quantification is usually required with respect to *all* the variables having secondary occurrences in a graph, and this can be implicit.

For example, in the state space exploration algorithm of Section 6.2.1, a set of states is represented by an MDG G representing a formula P of type $V \to Y$, where Y is the set of state variables, and V is a set of auxiliary abstract variables (representing values that have been read through data inputs, and initial values of data registers). Given an interpretation ψ, a state is an assignment ϕ to the variables in Y compatible with ψ. Such a state belongs to the set of states represented by P iff

$$\psi, \phi \models (\exists V)P. \tag{21}$$

In this sense, the variables in V are implicitly existentially quantified in G.

5.4 Relational product

As in [6] we provide a special purpose algorithm for the *relational product operation.* Our algorithm is a direct extension of the one of [6] to the more general MDG setting. An n-ary version of it is described in detail in Section A.3.

The relational product operation combines conjunction and existential quantification. Our algorithm takes the conjunction of a collection of MDGs having pairwise disjoint sets of abstract primary variables, and existentially quantifies with respect to some of the variables (abstract or concrete) that have primary occurrences in at least one of the graphs. (Notice that such variables would not have secondary occurrences in the conjunction of the graphs, if the conjunction were computed in a separate step.) In addition, it can rename some of the remaining primary variables. (This can be used to rename next-state variables to present-state variables.)

At this point our MDG package does not include separate algorithms for conjunction and existential quantification because these operations can be obtained as special cases of the relational product algorithm without substantial loss of efficiency. Conjunction is relational product with no variables being quantified or renamed, while existential quantification of an MDG G can be accomplished by taking the relational product of G with $\mathbf{T}$, with quantification but no renaming.[8]

[8]In fact, the algorithm defined in Appendix A.3 computes the relational of n MDGs, and n can be 1. Existential quantification can be accomplish by letting $n = 1$.

5.5 Pruning by subsumption

We have seen above how an MDG G of type $V \rightarrow Y$ representing a formula P can be used to denote the set of states ϕ such that $\psi, \phi \models (\exists V)P$ (for a given interpretation ψ). To check whether two MDGs G_1 and G_2 of type $V \rightarrow Y$, representing formulas P_1 and P_2, denote the same set of states, we need to check whether the formulas $(\exists V)P_1$ and $(\exists V)P_2$ are logically equivalent. The canonicity result of Section 4 tells us that we can test the logical equivalence of P_1 and P_2 by testing whether G_1 and G_2 are isomorphic; we take advantage of this in the usual way for combination verification (see Section 6.1). But the quantified formulas $(\exists V)P_1$ and $(\exists V)P_2$ may be logically equivalent even though G_1 and G_2 are non-isomorphic. Hence a test other than syntactic identity is needed to check whether two graphs denote the same set of states. The pruning-by-subsumption algorithm provides such a test. As an added bonus, it provides a form of *frontier-set simplification* for state space exploration, as we shall see in Section 6.2.1.

The algorithm takes as arguments two MDGs G and H of type $V \rightarrow Y$ (or, more generally, of type $V \rightsquigarrow Y$) representing formulas P and Q, and produces an MDG G' *derivable from* G *by pruning* according to the definition below, such that

$$\models P' \vee (\exists V)Q \Leftrightarrow P \vee (\exists V)Q \tag{22}$$

Definition 3 (Pruning) We say that an MDG G' is derivable from an MDG G by *pruning* iff it can be obtained by replacing some of the subgraphs of G with $\mathbf{F}$, then eliminating $\mathbf{F}$ and removing redundant nodes to obtain a well-formed MDG. To eliminate $\mathbf{F}$ is to find an internal node N and an edge E issuing from N that leads to a leaf node N' labeled $\mathbf{F}$, remove N' and E, turn N into a leaf node labeled $\mathbf{F}$ if E is the only edge issuing from N, and repeat the process until the graph becomes $\mathbf{F}$ or there are no leaf nodes labeled $\mathbf{F}$.

It is easy to see that, if P is of type $V \rightarrow Y$ and P' is derivable from P by pruning, then P' logically implies P and is either $\mathbf{F}$ or of type $V \rightarrow Y$.

Pruning amounts to removing some of the paths from P. The paths that are removed by the pruning-by-subsumption algorithm are those that are *subsumed* by Q according to the following definition.

Definition 4 (Path subsumption) Given two MDGs G and H of type $V \rightsquigarrow Y$, a path π of P is subsumed by a path π' of Q iff there exists a substitution θ with $dom(\theta) \subseteq V$ such that, if π'' is obtained by applying θ to the nodes and edges of π', then every node-edge pair of π'' is a node-edge pair of π. A path π of P is subsumed by the MDG Q if it is subsumed by a path π' of Q, or, more generally, if the addition of redundant nodes to π results in a collection of paths each of which is subsumed by some path of Q.

The algorithm can be used to detect set inclusion by checking whether G' is $\mathbf{F}$; if so, it follows from (22) that

$$\models (\exists V)P \Rightarrow (\exists V)Q, \tag{23}$$

which means that the set of states denoted by G is a subset of the set of states denoted by H. If H denotes the set of states found so far in a least fixpoint computation, and G the set of additional states found in the next iteration, $G' = \mathbf{F}$ means that the fixpoint has been reached.

A detailed description of the algorithm and a proof of soundness can be found in the appendix.

6 Verification

From now on we shall identify an MDG with the formula that it represents, when there is no risk of confusion.

6.1 Combinational verification

MDGs can be used in the same manner as BDDs to verify the functionality of a combinational circuit, or the transition and output relations of a sequential circuit. Consider a combinational or sequential circuit where the primary inputs are denoted by a set of variables X, the present values of the state registers (if any) by a set of variables Y, the primary outputs by a set of variables Z, and the next values of the state registers by a set of variables Y' in one-to-one correspondence with those of Y. The values of the variables in $Z \cup Y'$ are computed by a combinational circuit from the variables in $X \cup Y$. This combinational circuit is a collection of components connected by nets; each variable in $X \cup Y \cup Y' \cup Z$ denotes the value of a net, and additional internal variables denote the values of the internal nets. The values of data signals are represented by variables of abstract sorts, while the values of control signals are represented by variables of concrete sorts.

The behavior of a component with the set of inputs U and the set of outputs V is given by an MDG of type $U \to V$. Well-formedness condition 6 is satisfied if the abstract variables in U come before the abstract variables in V in the custom symbol order. Because there are no cycles in the combinational circuit, it is easy to find an ordering in which this is so for all the components in the circuit.

The desired value of each $z \in Z \cup Y'$ is specified by an MDG Q_z of type $(X \cup Y) \to \{z\}$. To check that the value produced by the circuit is as desired, we compute an MDG P_z of type $(X \cup Y) \to \{z\}$ by taking the conjunction of all the components in the cone of z, and removing any internal variables by existential quantification. This can be done by a single n-ary relational product computation, but it will typically be done more efficiently by breaking the computation into several steps and eliminating internal variables as soon as possible. Notice that no net can be the output of more than one component; hence the condition that the arguments of conjunction not share any abstract variables is automatically satisfied, without any need to take special precautions.

Once P_z has been computed, by the canonicity of MDGs, to check whether P_z and Q_z are logically equivalent it suffices to check whether they are isomorphic, which reduces to checking whether they are the same graph when a reduction table is used.

Recall that P_z is logically equivalent to Q_z iff, for *every* interpretation ψ,

$$\psi \models P_z \Leftrightarrow Q_z. \tag{24}$$

But (24) may hold for the intended interpretation ψ without holding for all ψ. In that case verification will fail even though the circuits are in fact equivalent, a false negative result. However, when data operations are viewed as *black boxes*, (24) is expected to hold for every ψ; hence, if the verification algorithm finds that this is not the case, there must be an error in the design. In this sense, we say that the verification method is applicable to designs where the data oparations can be viewed as black boxes. This black box requirement can be partially relaxed by the use of algebraic equations as rewrite rules, but this is beyond the scope of this paper. Similar considerations apply to the sequential verification algorithms described in the next section.

6.2 Sequential verification

We begin by giving an algorithm for proving that a given condition on the outputs of a sequential circuit holds for all the reachable states of the circuit. This algorithm is useful in its own right, and it is the basis for equivalence checking and verification against an algorithmic specification, which are briefly described in the sections that follow.

6.2.1 State space exploration

Consider a sequential circuit with primary inputs X, state variables Y, primary outputs Z, and next-state variables Y', and let W be a set of abstract variables, used for expressing the condition on the outputs. Assume, for simplicity, that these five sets of variables are pairwise disjoint.

The transition and output relations are given by two MDGs

$$\begin{array}{l} R_T : (X \cup Y) \to Y' \\ R_O : (X \cup Y) \to Z \end{array} \tag{25}$$

which can be computed as described above in Section 6.1.[9] The condition on the outputs is $(\exists W)C$, where C is an MDG

$$C : W \to Z. \tag{26}$$

The algorithm requires a supply of fresh abstract variables to denote values of data inputs and initial values of data registers. We assume that, for every $j \geq 1$, $U_j = \{u_{ij}\}_{1 \leq i \leq n}$ is a set of abstract variables disjoint from all U_k, $1 \leq k < j$, as well as from $X \cup Y \cup Y' \cup Z \cup W$, and in one-to-one correspondence with the set of abstract input variables $\{x_i\}_{1 \leq i \leq n} \subseteq X$. Each variable u_{ij} in U_j could be obtained, for example, by concatenating the name of the abstract input variable x_i with a symbol such as "#" not used in ordinary identifiers and with the digits of the decimal representation of j. For each $j \geq 1$, the MDG

$$I_j : U_j \to X \tag{27}$$

[9] Instead of these monolithic transition and output relations R_T and R_O, it is usually more efficient to use *partitioned* relations; see Section 7.

consists of a single path where the nodes are labeled by the x_i and the edges by the corresponding u_{ij}; it can be thought of as giving the values of the abstract input variables in clock cycle j.

We also use a set of abstract variables U_0, disjoint from all previously introduced sets of variables, and an MDG

$$I_0 : U_0 \rightarrow Y \tag{28}$$

to represent the set of initial states: an initial state is an assignment to the state variables that satisfies the formula $(\exists U_0) I_0$. For every $j \geq 0$ let

$$V_j = \bigcup_{i<j} (U_i). \tag{29}$$

To check whether the condition $(\exists W)C$ holds in all reachable states, we compute

$$\varepsilon = \mathbf{ReAn}(I_0, I_0, 1, R_T, R_O, C) \tag{30}$$

where **ReAn** (**Re**achability **An**alysis) is the following recursive algorithm.

Algorithm ReAn

$\varepsilon = \mathbf{ReAn}(P, Q, j, R_T, R_O, C)$

- Arguments: In addition to R_T, R_O, and C, described above, j is a positive integer and P, Q are two MDGs of type

  $$P, Q : V_j \rightarrow Y. \tag{31}$$

  (Q denotes the set of reachable states found so far, and P denotes the *frontier set* [6].)

- Result: $\varepsilon \in \{0, 1\}$; $\varepsilon = 0$ indicates that verification has succeeded, $\varepsilon = 1$ indicates that it has failed. If the initial call (30) returns $\varepsilon = 0$, then the condition $(\exists W)C$ holds for all outputs obtained in all reachable states for all input vectors.

Procedure:

1. (*Compute the outputs of the P states.*) Using the relational product operation, compute the MDG

$$O : V_{j+1} \rightarrow Z \tag{32}$$

such that

$$\models O \Leftrightarrow (\exists X)(\exists Y)(I_j \wedge P \wedge R_O). \tag{33}$$

O represents all possible output vectors in the states represented by P. (That is, the set of such outputs is the set of assignments to the variables Z that satisfy the formula $(\exists V_{j+1})O$.)

2. (*Check that they satisfy condition C.*) Using the pruning-by-subsumption algorithm, prune O to obtain an MDG

$$O' : V_{j+1} \rightarrow Z \tag{34}$$

such that

$$\models O' \vee (\exists W)C \Leftrightarrow O \vee (\exists W)C. \tag{35}$$

If $O' \neq \mathbf{F}$ then return $\varepsilon = 1$. Otherwise continue. (If $O' = \mathbf{F}$, then $\models (\exists V_{j+1})O \Rightarrow (\exists W)C$, and thus every output of a state represented by P satisfies the condition.)

3. (*Compute the states reachable in one step from the P states.*) Using the relational product algorithm, take the conjunction of I_j, P, and R_T, existentially quantify the variables in $X \cup Y$, and rename the variables in Y' to the corresponding variables in Y. The result is an MDG

$$P' : V_{j+1} \rightarrow Y \tag{36}$$

representing the states reachable in one step from the states represented by P.

4. (*Check if the P' states are already Q states.*) Using the pruning-by-subsumption algorithm, prune P' to obtain an MDG

$$P'' : V_{j+1} \rightarrow Y \tag{37}$$

such that

$$\models P'' \vee (\exists V_j)Q \Leftrightarrow P' \vee (\exists V_j)Q. \tag{38}$$

If $P'' = \mathbf{F}$, return $\varepsilon = 0$ (every state represented by P' is then among the states represented by Q; the fixpoint has been reached). Otherwise continue.

5. (*Simplify Q before taking the disjunction with P''.*) Using the pruning-by-subsumption algorithm, prune Q to obtain an MDG Q' such that

$$\models Q' \vee (\exists V_{j+1})P'' \Leftrightarrow Q \vee (\exists V_{j+1})P''. \tag{39}$$

6. (*Compute the new set of reachable states found so far.*) Using the disjunction algorithm, compute the MDG

$$Q'' : V_{j+1} \rightarrow Y \tag{40}$$

such that

$$\models Q'' \Leftrightarrow Q' \vee P''. \tag{41}$$

7. (*Recursive call.*) Compute and return

$$\varepsilon = \mathbf{ReAn}(P'', Q'', j+1, R_T, R_O, C). \tag{42}$$

Note that we use P'' rather than P' as the frontier set in the recursive call; P'' is derived from P' by pruning, and thus is smaller or equal in size to P'. Using P'' instead of P' can greatly facilitate the computation of the relational products in steps 1 and 3 of the recursive call. Using P'' is sound because (38) implies that every state covered by P' but not by P'' is already covered by Q. The use of P'' instead of P' is a form of frontier set simplification [6], but the simplification function that we use is not the same as used in prior work [15, 31].

Hence, in step 4 the pruning-by-subsumption algorithm plays a dual role: it is used both to check whether the fixpoint has been reached, and to simplify the frontier set.

Non-termination

As is the case for other state exploration methods based on abstract types, the above algorithm may not terminate. For example, if the the program counter of a microprocessor is represented by a variable pc of abstract type, the initial value of *pc* is given by a generic constant *zero*, and a function symbol *inc* denotes the incrementing of *pc* by a non-branching instruction, then a node of Q labeled by *pc* will have, in recursive invocations of the algorithms, an unbounded collection of edges *pc*, $inc(pc)$, $inc(inc(pc))$, etc., and the algorithm will not terminate.

As in [14, 26], this can be avoided by generalizing the initial state, using a variable x to denote the initial value of the program counter. A path of the frontier set MDG P where the *pc* edge is labeled $inc(x)$ will then be subsumed by a path of Q where the *pc* edge is labeled x; such paths will be pruned in step 4, and this will avoid the unbounded growth of Q causing non-termination. Generalizing the initial state amounts to supplying an invariant. In the general case, however, there is no guarantee that an invariant can be found to avoid non-termination.

The possibility of non-termination may seem a drawback with respect to BDD-based and other finite-state machine verification methods, which provide terminating algorithms. In practice, however, finite-state verification algorithms often take too long, or exhaust memory; the fact that they are theoretically terminating is then of no practical consequence. And verification methods based on abstract types can be expected to terminate for important classes of problems, including the verification of general purpose microprocessors [14, 24, 25].

6.2.2 Verification of behavioral equivalence

To verify that two circuits have same input/output behavior it suffices to derive the transition and output relations of their product machine, and then check that corresponding outputs of the product machine are equal in all reachable states using the above state exploration algorithm. (The product machine is obtained by placing the circuits to be compared side by side and feeding them the same inputs.)

6.2.3 Verification against an algorithmic specification

An algorithmic specification is a sequential program, such as a VHDL process, that specifies the operations to be performed by the hardware [13, 29]. Such a specification can be viewed as a micro-state machine implementing a macro-state machine. A transition of the micro-state machine consists of the execution of an instruction of the sequential program, while a transition of the macro-state machine is realized by executing the process until it is suspended. The transition and output relations of the macro-state machine can be derived from those of the micro-state machine by a fixpoint computation, and the macro-state machine can then be compared to the state machine derived from the circuit as described in Section 6.2.2.

7 Implementation and initial experimental results

We have implemented in Prolog an MDG package that includes algorithms for disjunction, relational product, and pruning by subsumption. We have chosen Prolog only because it permits fast prototyping and experimentation with various versions of the algorithms. It is well known that very high-level languages such as Prolog or LISP are not well-suited for implementing BDD algorithms efficiently; for this reason, efficient BDD packages are implemented in C. The same is true for MDG algorithms, and we conjecture that a C implementation would yield much better run-times.

Using the MDG package we have implemented the algorithms for combinational verification, reachability analysis, and equivalence checking described in Sections 6.1, 6.2.1, and 6.2.2, as well as a counterexample facility that we plan to describe elsewhere.

As our initial case study we tried the combinational verification and state space exploration algorithms described in Section 6 by comparing two identical implementations of the Tamarack-3 microprocessor[22, 14] to each other. To try out the combinational verification algorithm, we compared the transition and output relations of the two implementations; the CPU time for this was 27 seconds on a Sun SPARCserver 630MB. To try out the state space exploration algorithm we formed the product machine of the two implementations, ignoring the known correspondence between their registers. Then, as in [14] we checked that the outputs *dreq* and *wmem* have the same value in both circuits in all reachable states, and that *addr* and *dataout* have the same value when the values are relevant. Instead of using a monolithic transition relation as described in Section 6.2.1, we used a *partitioned transition* *relation* [6].[10] We also used the partition ordering heuristic of [18]. The total CPU time was 60 seconds, including 35 seconds for compiling the circuit and computing the partitioned transition and output relations.

The CPU time for equivalence checking with our MDG package is greater than the time reported in [14] even though we use a faster machine. This is not surprising: just like ordinary explicit enumeration can be faster in some cases than BDD-based implicit enumeration [19], the *abstract explicit enumeration* method used in [14] can be faster than the MDG-based *abstract implicit enumeration* method that we have described. Yet, implicit enumeration should be faster on larger examples in most cases. Also, moving from Prolog to C should yield a greater improvement for the MDG method than for the method of [14], since the latter takes advantage of the built-in facilities of Prolog for efficient backtracking and unification, while the MDG package does not.

8 Conclusion

We have described a new kind of decision graphs, MDGs, which incorporate abstract sorts and uninterpreted function symbols, and where a data signal can be represented by a single variable of abstract sort rather than by 32 or 64 boolean variables. We have described a basic package of algorithms for computing with MDGs, and shown how they can be used for combinational verification, state space exploration, and verification of behavioral equivalence of sequential circuits. We have provided initial experimental results obtained with a prototype MDG package implemented in Prolog.

Previous work has shown that the use of abstract types for formal verification can achieve dramatic results, by raising the level of abstraction at which the problem is stated. The contribution of MDGs beyond the use of abstract types is that they make it possible to use some of the BDD techniques that have been successful at the boolean level at a higher level of abstraction. In

[10] Our n-ary relational product algorithm allows for two levels of partitioning: the transition relation is represented by a collection of MDGs as usual, but then each relational product in the next state computation can involve several of those MDGs. The results described below have been obtained by using one MDG for each next-state variable, and then grouping those MDGs into two sets, so that only two relational products are computes for each next-state computation. In the future we plan to conduct experiments to evaluate the trade-offs allowed by these two levels of partitioning.

particular, the experiments that we have reported are the first, to our knowledge, in which abstract implicit enumeration has been used for state space exploration; by abstract implicit enumeration we mean the use of decision graphs incorporating variables of abstract types to represent sets of states.

We believe that it is possible to use conditional rewriting techniques to exploit partial knowledge about the uninterpreted function symbols occurring in MDGs, and we hope to be able to provide MDG rewriting algorithms in the near future.

MDGs are a computational tool rather than a verification method. They open the way to the development of new techniques for the verification of circuit and system designs at higher levels of abstraction.

References

[1] F. Van Aelten, S. Y. Liao, J. Allen, and S. Devadas. Automatic generation and verification of sufficient correctness properties for synchronous processors. In *International Conference on Computer-Aided Design*, 1992.

[2] S. Bose and A. L. Fisher. Automatic verification of synchronous circuits using symbolic logic simulation and temporal logic. In *Proceedings of the IMEC–IFIP Workshop on Applied Formal Methods for Correct VLSI Design*, 1989.

[3] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a bdd package. In *27th ACM/IEEE Design Automation Conference*, 1990.

[4] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[5] R. E. Bryant, D. L. Beatty, and C.-J. H. Seger. Formal hardware verification by symbolic ternary trajectory evaluation. In *28th ACM/IEEE Design Automation Conference*, 1991.

[6] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design*, 13(4):401–424, April 1994.

[7] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Proc. Work. on Computer-Aided Verification*, 1994.

[8] A. J. Camilleri, M. J. C. Gordon, and T. F. Melham. Hardware verification using higher-order logic. In D. Borrione, editor, *Proceedings of the IFIP WG 10.2 Working Conference: From H.D.L. Descriptions to Guaranteed Correct Circuit Designs, Grenoble, September 1986.* North-Holland, 1987.

[9] H. Cho, G. Hachtel, S.-W. Jeong, B. Plessier, E. Schwarz, and F. Somenzi. ATPG aspects of FSM verification. In *International Conference on Computer-Aided Design*, 1990.

[10] L. Claesen, F. Proesmans, E. Verlind, and H. De Man. SFG-Tracing: A methodology for the automatic verification of MOS transistor level implementations from high level behavioral specifications. In P. A. Subrahmanyam, editor, *International Workshop on Formal Methods in VLSI Design, Miami, Florida*, January 1991.

[11] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proc. 19th ACM Symp. on Principles of Programming Languages*, January 1992.

[12] Francisco Corella. What holds in a context? *Journal of Automated Reasoning*, 10:79–93, 1993.

[13] Francisco Corella. Automated high-level verification against clocked algorithmic specifications. In *Proceedings of CHDL 93*, April 1993.

[14] Francisco Corella. Automated verification of behavioral equivalence for microprocessors. *IEEE Transactions on Computers*, 43(1):115–117, January 1994.

[15] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems.* Springer Verlag, 1989. Lecture Notes in Computer Science, vol. 407.

[16] O. Coudert and J. C. Madre. A unified framework for the formal verification of sequential circuits. In *International Conference on Computer-Aided Design*, 1990.

[17] M. Fujita. RTL design verification by making use of datapath information. In *IEEE International Conference on Computer Design*, 1992.

[18] D. Geist and I. Beer. Automatic verification of pipelined microprocessor control. In *Proc. Work. on Computer-Aided Verification*, 1994.

[19] A. J. Hu and D. L. Dill. Reducing BDD size by exploiting functional dependencies. In *30th ACM/IEEE Design Automation Conference*, pages 266–271, 1993.

[20] Warren A. Hunt. *FM8501: A Verified Microprocessor.* PhD thesis, University of Texas, Austin, 1985.

[21] J. Joyce, G. Birtwistle, and M. Gordon. Proving a computer correct in higher order logic. Technical Report 100, University of Cambridge, Computer Laboratory, December 1986.

[22] Jeffrey J. Joyce. Multi-level verification of microprocessor-based systems. Technical Report 195, University of Cambridge, Computer Laboratory, May 1990. PhD thesis.

[23] R. P. Kurshan. Analysis of discrete event coordination. In J. W. de Bakker, W.-P. de Roewer, and G. Rozenberg, editors, *Proceedings of the REX Workshop on Stepwise Refinement of Distributed Systems.* Springer Verlag, 1989. LNCS 430.

[24] M. Langevin and E. Cerny. Verification of processor-like circuits. In P. Prinetto and E. Camurati, editors, *Proc. Work. on Correct Hardware Design Methodologies.* North-Holland, June 1991.

[25] M. Langevin and E. Cerny. Comparing generic state machines. In K. G. Larsen and A. Skou, editors, *Proc. Work. on Computer-Aided Verification*. Springer-Verlag LNCS 575, July 1991.

[26] M. Langevin and E. Cerny. An approach for behavioral verification of circuit designs. In *Proc. Canadian Conf. on VLSI*, October 1992.

[27] M. Langevin and E. Cerny. An extended OBDD representation for extended FSMs. In *Proc. of EDAC-ETC-EUROASIC*, 1994.

[28] David E. Long. *Model Checking, Abstraction, and Compositional Verification*. PhD thesis, Carnegie Mellon University, 1993.

[29] M. Payer, T. Filkorn, and P. Warkentin. Symbolic verification of sequential circuits synthesized with CALLAS. In *International Workshop on High-Level Synthesis*, Laguna Nigel, California, 1992.

[30] A. Srinivasan, T. Kam, S. Malik, and R. K. Brayton. Algorithms for discrete function manipulation. In *ACM/IEEE 27th Design Automation Conference*, 1990.

[31] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDDs. In *International Conference on Computer-Aided Design*, 1990.

[32] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, January 1986.

A Details of the MDG algorithms

In this appendix we provide a detailed description of the algorithms for disjunction, relational product and pruning-by-subsumption.[11] There is no difficulty in verifying the soundness of the algorithms, except perhaps for the pruning algorithm, for which a proof of soundness is given in Section A.5.

[11] The descriptions given here do not always match exactly the implementations of the algorithms in the prototype MDG package. The differences, however, are minor.

A.1 Technical preliminaries

If an MDG G represents a formula P, we shall use P to refer both to the formula and to the graph when there is no risk of confusion. We shall also sometimes identify nodes and edges with their labels.

We shall refer to the edges that issue from the root of an MDG as the *root edges*, and to the subgraphs that they lead to as the *immediate subgraphs*.

A *full node* is an internal node N labeled by a variable or cross-term of concrete sort α, such that every individual constant in the enumeration of α appears as a label of an edge issuing from N. (A redundant node is a special case of a full node, where all the edges issuing from the node lead to the same subgraph.)

A *segment* σ of an MDG G is a maximal sequence of consecutive node-edge pairs where all the node labels have same top symbol; we shall refer to this common top symbol as the *top symbol of the segment*. A segment σ can thus consist of a single node N, labeled by a variable, and an edge issuing from N, or of a sequence of node-edge pairs where the nodes are labeled by cross-terms with the same cross-operator. If $[(A_i, B_i)]_{1 \leq i \leq n}$ is the sequence of the node-edge pairs in σ, we shall identify σ with the expression $\bigwedge_{1 \leq i \leq n}(A_i = B_i)$. Note that σ could also be identified with the graph obtained by adding a leaf node labeled $\mathbf{T}$ at the end of the sequence of consecutive nodes and edges. Such a graph G_σ is a well-formed MDG; we shall say that σ is of type $X \rightsquigarrow Y$ to mean that G_σ is of type $X \rightsquigarrow Y$.

We call *root segments* those that begin at the root.

A *substitution* θ with a set of variables Y as its domain is, as usual, a function that maps every $y \in Y$ to a term of the same sort as y. We shall write $A \cdot \theta$ for the result of *applying* the substitution θ to A, where A can be a term, a formula, a variable, a set of variables, or so on; for example, if x is a variable *not* in the domain of θ, $\theta(x)$ is undefined, but $x \cdot \theta$ is x itself.

A *renaming substitution* is one that maps variables to variables in a one-to-one manner. We say that a renaming substitution θ *preserves an ordering relation* $\prec$ over a set of symbols S iff the function $f : x \in S \mapsto x \cdot \eta$ is an ordering isomorphism from $(S, \prec)$ onto $(f(S), \prec)$. We shall use this notion in the case where $\prec$ is the custom symbol ordering, S is a set of variables and cross-operators, and the domain and range of η consist of variables that participate in the symbol ordering; in that case η preserves the symbol ordering iff

$$(\forall x, x' \in S)(x \prec x' \Rightarrow x \cdot \eta \prec x' \cdot \eta). \tag{43}$$

Besides renaming substitutions we shall only consider substitutions θ that map abstract variables to reduced terms. That is, every $y \in dom(\theta)$ is abstract, and every $\theta(y)$ is a reduced term. Such a substitution can be identified with a linear MDG G_θ with node-edge pairs $[(y_i, \theta(y_i)]_{1 \leq i \leq n}$, where y_i, $1 \leq i \leq n$, are the variables of Y in custom order, and with leaf node $\mathbf{T}$. It can also be identified with the formula $\bigwedge_{1 \leq i \leq n}(y_i = \theta(y_i))$. The MDG G_θ is not always well-formed. For G_θ to be well-formed every variable x that occurs in some $\theta(y)$, $y \in Y$, must come before the corresponding y in custom symbol order, and the set X of such x must be disjoint from Y. When this is the case, we say that θ itself is well-formed; and we say that θ is of type $X' \to Y'$ iff G_θ is of type $X' \to Y'$: this means that $X \subseteq X'$, $Y \subseteq Y'$, and $Y_{\mathsf{abs}} = (Y')_{\mathsf{abs}}$.

The MDG algorithms use a common *reduction table* and separate *result tables*, all implemented as hash tables. Each MDG has a unique identifier. The reduction table has an entry for each MDG that maps the label of the root, the sequence of labels of the root edges, and the sequence of identifiers of the immediate subgraphs to the identifier of the graph. The ID of the graph can be used as an index into a (variable-sized) array to retrieve the actual graph. In the descriptions of the algorithms, when we say that we construct a graph with given root label, root edge labels, and immediate subgraphs, it should always be understood that the reduction table is looked up; if an entry is found, the graph is retrieved from the array; if no entry is found, a new graph is assembled and added to the array, and a new entry is added to the reduction table. The result tables are used to cache the results computed by the algorithm; their use will be explicitly described for each algorithm.

We define a total ordering on the set of MDGs other than $\mathbf{T}$ and $\mathbf{F}$, the *graph ordering*, as follows: G comes before G' iff the root label of G comes before the root label of G' in node-label ordering, or G and G' have same root label and the identifier of G is less than the identifier of G' (identifiers are natural numbers). When sorting lists of MDGs, the ordering used will always be graph order.

A.2 Disjunction

$R = \mathbf{Disj}(\{P_i\}_{1 \le i \le n})$

- Argument: A set $S = \{P_i\}_{1 \le i \le n}$ of MDGs P_i. Each P_i other than $\mathbf{T}$ or $\mathbf{F}$ is an MDG of type $U_i \rightarrow Y$. (Note that all the P_i have the same set Y_{abs} of primary abstract variables.)

- Result: an MDG R such that

$$\models R \Leftrightarrow (\bigvee_{1 \le i \le n} P_i) \tag{44}$$

  R can be $\mathbf{F}$, $\mathbf{T}$, or an MDG of type $V \rightarrow Y$, where V is the union of the sets of variables U_i.

If any of the P_i is $\mathbf{T}$, return $R = \mathbf{T}$. If $n = 0$ or all the P_i are $\mathbf{F}$, return $R = \mathbf{F}$. Otherwise sort the P_i other than $\mathbf{F}$ and remove duplicates, thus obtaining a list $[P'_i]_{1 \le i \le n'}$.

Check if the disjunction of the P'_i, $1 \le i \le n'$, has already been computed. The result table for disjunction maps a list of graph IDs to a graph ID. To check if the disjunction has already been computed, extract a list of graph IDs from the list $[P'_i]_{1 \le i \le n'}$, compute a hash key from the list of IDs, and look up the result table. If an entry is found, use the result ID to look up the graph array, and return the corresponding graph as the result of the disjunction.

If the disjunction has not been computed previously, let A be the smallest of the root labels of the P'_i, $1 \le i \le n'$, in node-label order. Since the P'_i are sorted in graph order, those whose root label is A appear at the front of the list. If A is an abstract variable, then it is the root label of *all* the P'_i.

For each i, $1 \le i \le n'$, let $L_i = [(B_{ij}, Q_{ij})]_{1 \le j \le m_i}$ be a list of pairs defined as follows:

1. If the root of P'_i is A, then the B_{ij}, $1 \le j \le m_i$ are the root edges of P'_i (in standard term order, as they appear in P'_i), and the Q_{ij}, $1 \le j \le m_i$ are the immediate subgraphs that those edges lead to.

2. If the root of P'_i comes after A in node-label order, it must be the case that A is a concrete variable or a cross-term. Let α be the sort of A. Then the B_{ij}, $1 \le j \le m_j$ are the individual constants in the enumeration of α, in standard term order, and $Q_{ij} = P'_i$ for all j.

Merge the L_i, $1 \leq i \leq n'$, to obtain a list of pairs $[(B'_k, S'_k)]_{1\leq k\leq l}$, where the B'_k are the B_{ij}, $1 \leq i \leq n'$, $1 \leq j \leq m_i$, arranged in standard term order without duplicates, and S'_k is the set of P'_{ij} for which the corresponding B_{ij} is B'_k. For each k, $1 \leq k \leq l$, recursively compute $Q'_k = \mathbf{Disj}(S'_k)$. If A is of concrete sort α, if all the Q'_k are the same, and if the B'_k comprise all the individual constants in the enumeration of α, then return $R = Q'_1 = \ldots = Q'_l$. Otherwise construct an MDG R with root A, root edges $[B'_k]_{1\leq k\leq l}$ and immediate subgraphs $[Q'_k]_{1\leq k\leq l}$ and return it as R. Add an entry to the result table and return R as the result.

A.3 Relational product

$R = \mathbf{RelP}(\{P_i\}_{1\leq i\leq n}, V, \eta)$

- Arguments: A (possibly empty) set $S = \{P_i\}_{1\leq i\leq n}$, $n \geq 0$, of MDGs P_i, each being either $\mathbf{T}$, or $\mathbf{F}$, or of type $X_i \rightarrow Y_i$, a set of variables V, and a renaming substitution η. Let $X = \bigcup_{1\leq i\leq n} X_i$ and $Y = \bigcup_{1\leq i\leq n} Y_i$. Conditions:

  1. The $(Y_i)_{\mathsf{abs}}$ are pairwise disjoint.
  2. $V \subseteq Y$.
  3. The domain of η is a subset of $Y \setminus V$, and η preserves the custom ordering over the union of $(Y \setminus V) \cup (X \setminus Y)$ and the set of cross-operators.

- Result: an MDG R that can be $\mathbf{F}$, $\mathbf{T}$, or an MDG of type $(X \setminus Y) \rightarrow ((Y \setminus V) \cdot \eta)$ such that

$$\models R \cdot \eta^{-1} \Leftrightarrow (\exists V)(\bigwedge_{1\leq i\leq n} P_i). \tag{45}$$

  (Note that the set of primary abstract variables of R is the set obtained by taking all the primary abstract variables of the P_i, removing those that are elements of V, and, among the remaining ones, renaming those that are in $dom(\eta)$.)

If any of the P_i is $\mathbf{F}$, return $R = \mathbf{F}$. If $n = 0$ or all the P_i are $\mathbf{T}$, return $R = \mathbf{T}$. Otherwise sort the P_i other than $\mathbf{T}$ into a list L, then compute and return

$$R = \mathbf{RelP'}(L, V, \eta, \emptyset). \tag{46}$$

Note that we have not mentioned a result table. The result table for relational product is used by the auxiliary algorithms **RelP'** and **RelP"** rather than by **RelP** itself.

Auxiliary Recursive Algorithm RelP'

$R = \mathbf{RelP'}([P_i]_{1 \le i \le n}, V, \eta, \theta)$.

- Arguments: A list $L = [P_i]_{1 \le i \le n}$, $n \ge 0$, of MDGs $P_i \ne \mathbf{T}, \mathbf{F}$ of type $X_i \to Y_i$, a set of variables V, a renaming substitution η, and a well-formed substitution θ of type $X' \to Y'$. Let $X = \bigcup_{1 \le i \le n} X_i$, $Y = \bigcup_{1 \le i \le n} Y_i$, $X'' = X \cup X'$, and $Y'' = Y \cup Y'$. Conditions:

  1. The $(Y_i)_{\mathsf{abs}}$ are pairwise disjoint.
  2. Every variable in Y comes before every variable in Y' in custom symbol order.
  3. $V \subseteq Y''$.
  4. The domain of η is a subset of $Y'' \setminus V$, and η preserves the custom symbol ordering over the union of $(Y'' \setminus V) \cup (X'' \setminus Y'')$ and the set of cross-operators.

- Result: an MDG R that can be $\mathbf{F}$, $\mathbf{T}$, or an MDG of type $(X'' \setminus Y'') \to ((Y'' \setminus V) \cdot \eta)$ such that

$$\models \theta \Rightarrow (R \cdot \eta^{-1} \Leftrightarrow (\exists V)(\bigwedge_{1 \le i \le n} P_i)). \tag{47}$$

If $n = 0$, return $R = \mathbf{T}$. Otherwise, check the result table to see if the result has already been computed. The result table maps the list of IDs of the graphs P_i, a *task descriptor* consisting of an abbreviated representation of the pair (V, η), and the substitution θ, to the ID of the result R. Extract the list of graph IDs from the list of graphs $L = [P_i]_{1 \le i \le n}$, compute a hash key from the list of IDs, the task descriptor, and the substitution θ, and look up the result table. If an entry is found, use the graph ID that it supplies to retrieve the previously computed result from the graph array. Otherwise proceed as follows.

Let m be the number of graphs P_i at the front of the list L that have same top symbol. (The top symbol of an MDG is the top symbol of the label of its root; we shall refer to the top symbol of the MDGs at the front of the list as the *front top symbol.*) Consider the following cases:

1. The front top symbol is a variable y of concrete type α. Then for $1 \leq i \leq m$, P_i has y as root label, and individual constants in the enumeration of α as labels of the root edges. Let $[b_j]_{1 \leq j \leq l}$ be the list of constants that appear as root edge labels in all the P_i, arranged in standard order. If the list is empty ($l = 0$), return $R = \mathbf{F}$. Otherwise, for $1 \leq j \leq l$ and $1 \leq i \leq m$, let Q_{ij} be the immediate subgraph of P_i corresponding to the edge labeled b_j.

   For each j, construct a (possibly empty) sorted list L'_j, starting with the (possibly empty) tail $[P_k]_{m<k\leq n}$ of L and inserting in turn each $Q_{ij} \neq \mathbf{T}$ (if any) where it belongs. Compute the restriction θ'_j of θ to the set of variables that have secondary occurrences in the graphs that comprise L'_j. And recursively compute:

   $$Q'_j = \textbf{RelP'}(L'_j, V, \eta, \theta'_j). \tag{48}$$

   Then compute the result R as follows:

   (a) If $Q'_j = \mathbf{F}$ for $1 \leq j \leq l$, let $R = \mathbf{F}$.

   (b) Otherwise, if $y \notin V$, then:

      i. If the Q'_j are all the same MDG for $1 \leq j \leq l$, and if the b_j comprise all the individual constants in the enumeration of α, let R be that MDG.

      ii. Otherwise let R be the MDG with root labeled $y \cdot \eta$ and one root edge labeled b_j leading to Q'_j for each j such that $Q'_j \neq \mathbf{F}$.

   (c) Otherwise, use algorithm **Disj** to compute the disjunction of the $Q'_j \neq \mathbf{F}$, and let the result be R.

   Return R after entering it in the result table.

2. The front top symbol is an abstract variable y. Then, since the $(Y_i)_{\mathrm{abs}}$ are pairwise disjoint, it must be that $m = 1$. P_1 has root label y, and root edges labeled by terms B_j, $1 \leq j \leq l$. Compute $B_j \cdot \theta$ for each j. Form the list $[B_j \cdot \theta]_{1 \leq j \leq l}$, then sort it and remove duplicates to obtain a list $[B'_k]_{1 \leq k \leq h}$ ($h \leq l$).

For each k, using algorithm **Disj**, compute the disjunction Q_{1k} of the immediate subgraphs of P_1 corresponding to the B_j such that $B_j \cdot \theta = B'_k$; if $Q_{1k} = \mathbf{T}$, let L'_k be the (possibly empty) tail $[P_i]_{2\leq i\leq n}$ of L, otherwise let L'_k be the sorted list obtained by inserting Q_{1k} at its proper position in $[P_i]_{2\leq i\leq n}$. Form the substitution $\theta'_k = \theta \cup \{(y, B'_k)\}$, then compute the restriction θ''_k of θ'_k to the set of variables that have secondary occurrences in the graphs that comprise L'_k. And recursively compute

$$Q'_k = \mathbf{RelP'}(L'_k, V, \eta, \theta''_k). \tag{49}$$

Then compute the result R as follows:

(a) If $Q'_k = \mathbf{F}$ for $1 \leq k \leq h$, let $R = \mathbf{F}$.

(b) Otherwise, if $y \notin V$, let R be the MDG with root labeled $y \cdot \eta$ and one root edge labeled B'_k leading to Q'_k for each k such that $Q'_k \neq \mathbf{F}$.

(c) Otherwise use algorithm **Disj** to compute the disjunction of the $Q'_k \neq \mathbf{F}$, and let the result be R.

Return R after entering it in the result table.

3. The front top symbol is a cross-op f. (The root labels of the P_i for $1 \leq i \leq m$ are cross-terms with cross-op f.) In that case compute $P'_i = \mathbf{ST}(P_i, \theta)$ for $1 \leq i \leq m$, using the algorithm **ST** described below. If any of the P'_i is $\mathbf{F}$, return $R = \mathbf{F}$. Otherwise construct a (possibly empty) sorted list L', starting with the (possibly empty) tail $[P_j]_{m<j\leq n}$ of L and inserting in turn each $P'_i \neq \mathbf{T}$ (if any) where it belongs. Compute the restriction θ' of θ to the set of variables that have secondary occurrences in the graphs that comprise L'. Then use the algorithm **RelP"** described below to compute

$$R = \mathbf{RelP''}(L', V, \eta, \theta, f). \tag{50}$$

Check again if there is an entry in the **RelP'** result table for the current computation (one may have been added by **RelP"**, which shares the same result table); if not, add one with R as result. Finally, return R as the result.

Auxiliary algorithm ST ("Substitution at the Top")

$R = \mathbf{ST}(P, \theta)$

- Arguments: An MDG P, of type $X \rightarrow Y$, having a cross-op f as top symbol, and a well-formed substitution θ, of type $X' \rightarrow Y'$, such that $Y' < Y$.

- Result: An MDG R such that

  - $\models \theta \Rightarrow (P \Leftrightarrow R)$.
  - R is $\mathbf{F}$, $\mathbf{T}$, or an MDG of type $(X \cup X') \rightarrow Y$, with f as top symbol and no occurrences of Y' variables in the root segments.

For each root segment σ of P, apply the substitution θ to the node labels, then sort the node-edge pairs so that node labels appear again in node-label order; there may now be duplicate node-edge pairs in σ; remove them. The resulting segment may be contradictory, containing pairs with same node label but different individual constants as edge labels.

Let σ_i, $1 \leq i \leq n$, be the original segments of P that give rise to non-contradictory segments after substitution. For each i, $1 \leq i \leq n$, let P_i' be the subgraph of P that σ_i leads to, and let σ_i' be the result of applying θ to σ_i, sorting and removing duplicates. For each i, construct the MDG Q_i consisting of a single root segment σ_i' leading to a subgraph that coincides with P_i'. Then compute and return

$$R = \mathbf{Disj}([Q_i]_{1 \leq i \leq n}). \tag{51}$$

Auxiliary Recursive Algorithm RelP"

$R = \mathbf{RelP"}([P_i]_{1 \leq i \leq n}, V, \eta, \theta, f)$.

- Arguments: $L = [P_i]_{1 \leq i \leq n}$, V, η, and θ are as in **RelP'**, with P_i of type $X_i \rightarrow Y_i$, $X = \bigcup_{1 \leq i \leq n} X_i$, $Y = \bigcup_{1 \leq i \leq n} Y_i$, and θ of type $X' \rightarrow Y'$. The fourth argument f is a cross-op with target-type α. Let $X'' = X \cup X'$, $Y'' \cup Y'$. The conditions on the arguments are as follows:

1...4: as in **RelP'**.

5. When $n > 0$, if s is the front top symbol of L, $f \leq s$ in custom order.

6. Cross-terms with cross-op f in L have no occurrences of variables in Y'.

- Result: as in **RelP'**, an MDG R, which can be **F**, **T**, or of type $(X'' \setminus Y'') \to ((Y'' \setminus V) \cdot \eta)$ such that

$$\models \theta \Rightarrow (R \cdot \eta^{-1} \Leftrightarrow (\exists V)(\bigwedge_{1 \leq i \leq n} P_i)). \tag{52}$$

If $n = 0$, return $R = \mathbf{T}$. Otherwise, if $f < s$, compute and return

$$R = \mathbf{RelP'}(L, V, \eta, \theta). \tag{53}$$

Otherwise, check if the result has already been computed. **RelP"** and **RelP'** share the same result table. Extract a list of graph IDs from the MDGs that comprise L and use it to look up the result table as described above for **RelP'**. If an entry is found, use the resulting ID to retrieve the result from the graph array. Otherwise proceed as follows.

Let m be the number of P_i with same root label A at the front of L; A is a cross-term with cross-op f. (Note that there may be MDGs in L after the first m ones whose root label is a also a cross-term with cross-op f, a cross-term that comes after A in node-label order.) Let $[b_j]_{1 \leq j \leq l}$ be the list of constants that appear as root edge labels in all the P_i, arranged in standard order. If the list is empty ($l = 0$), return $R = \mathbf{F}$. Otherwise, for $1 \leq j \leq l$ and $1 \leq i \leq m$, let Q_{ij} be the immediate subgraph of P_i corresponding to the edge labeled b_j.

For each j, construct a sorted list L'_j, starting with the (possibly empty) tail $[P_k]_{m<k\leq n}$ of L and inserting each Q_{ij} where it belongs. Compute the restriction θ' of θ to the set of variables that have secondary occurrences in the graphs that comprise L'_j. And recursively compute

$$Q'_j = \mathbf{RelP"}(L'_j, V, \eta, \theta', f). \tag{54}$$

Then compute the result R as follows:

1. If $Q'_j = \mathbf{F}$ for $1 \leq j \leq l$, let $R = \mathbf{F}$.

2. If the Q'_j are all the same MDG for $1 \leq j \leq l$, and if the b_j comprise all the individual constants in the enumeration of α, let R be that MDG.

3. Otherwise let R be the MDG with root labeled A and one root edge labeled b_j leading to Q'_j for each j such that $Q'_j \neq \mathbf{F}$.

Return R after entering it in the result table.

A.4 Algorithm PbyS

$P' = \mathbf{PbyS}(P, Q)$

- Arguments: Two MDGs P and Q of type $V \rightsquigarrow Y$, where V contains only abstract variables that do not participate in the custom symbol ordering; P and Q can both be $\mathbf{T}$ or $\mathbf{F}$.

- Result: An MDG P', derivable from P by pruning, such that:

$$\models P \vee (\exists V)Q \Leftrightarrow P' \vee (\exists V)Q \tag{55}$$

  Since P' is derivable from P by pruning, it is, like P, of type $V \rightsquigarrow Y$. Moreover, if P is of type $V \rightarrow Y$ (i.e. if all the variables in Y_{abs} occur along the paths of P), then P' is also of type $V \rightarrow Y$ when it is not $\mathbf{F}$.

If $Q = \mathbf{F}$ return $P' = P$. If $Q = \mathbf{T}$ or $P = \mathbf{F}$ return $P' = \mathbf{F}$. Otherwise compute and return

$$P' = \mathbf{PbyS'}(P, [(Q, \emptyset)]). \tag{56}$$

The result table for pruning by subsumption is used by the auxiliary algorithm **PbyS'** rather than by **PbyS** itself.

Auxiliary algorithm PbyS'

$P' = \mathbf{PbyS'}(P, [(Q_i, \theta_i)]_{1 \leq i \leq n})$

- Arguments: An MDG P of type $V \rightsquigarrow Y$ other than $\mathbf{F}$, and a list of graph-substitutions pairs (Q_i, θ_i), $1 \leq i \leq n$, possibly empty ($n \geq 0$), where each Q_i is an MDG of type $V \rightsquigarrow Y$ other than $\mathbf{T}$ or $\mathbf{F}$, and each θ_i is a substitution that maps some of the variables in V to concretely-reduced terms containing only variables that are elements of V. The list of graph-substitution pairs is sorted and without duplicates. The sorting criterion does not matter: the purpose of keeping the list sorted is only to increase the hit rate in the result table.

- Result: An MDG P' of type $V \rightsquigarrow Y$, derivable from P by pruning, such that
$$\models P \vee \left(\bigwedge_{1 \leq i \leq n} (Q_i * \theta_i) \right) \Leftrightarrow P' \vee \left(\bigwedge_{1 \leq i \leq n} (Q_i * \theta_i) \right) \tag{57}$$
where a 0-ary conjunction ($n = 0$) means $\mathbf{T}$ by definition, and the notation $Q * \theta$ is defined as follows, for an arbitrary MDG Q of type $V \rightsquigarrow Y$ and an arbitrary substitution θ with domain $V_1 \subseteq V$: let $V_2 = V \setminus V_1$; let W be a set of abstract variables disjoint from $V \cup Y$; let $\eta : V_2 \mapsto W$ be a renaming substitution; and let $\theta' = \theta \cup \eta$; then $Q * \theta$ stands for
$$(\exists W)(Q \cdot \theta') \tag{58}$$
(defined up to renaming of bound variables).

If $P = \mathbf{T}$ then: if $n = 0$, return $P' = \mathbf{F}$, otherwise return $P' = \mathbf{T}$. If $P \neq \mathbf{T}$, check if the result has already been computed. Extract the IDs of P and the Q_i, compute a hash key from those graph IDs and the substitutions θ_i, and look up the result table. If an entry is found, use the ID that it supplies to retrieve the previously computed result from the graph array. Otherwise proceed as follows.

For each segment σ issuing from N, R being the subgraph that σ leads to, compute
$$R' = \mathbf{PbyS"}(\sigma, R, [(Q_i, \theta_i)]_{1 \leq i \leq n}) \tag{59}$$
and replace R with R'. From the resulting graph, obtain P' by eliminating $\mathbf{F}$ if necessary (i.e. if $R' = \mathbf{F}$) and removing redundant nodes. Return P' after entering it in the result table.

Auxiliary algorithm PbyS"

$R' = \mathbf{PbyS"}(\sigma, R, [(Q_i, \theta_i)]_{1 \leq i \leq n})$

- Arguments: σ is a segment of type $V \rightsquigarrow Y$, R is an MDG of type $V \rightsquigarrow Y$ other than $\mathbf{F}$, the list $[(Q_i, \theta_i)]_{1 \leq i \leq n}$ is as before, with $n > 0$.

- Result: a $V \rightsquigarrow Y$ graph R', derivable from R by pruning, such that
$$\models (\sigma \wedge R) \vee \left(\bigwedge_{1 \leq i \leq n} (Q_i * \theta_i) \right) \Leftrightarrow (\sigma \wedge R') \vee \left(\bigwedge_{1 \leq i \leq n} (Q_i * \theta_i) \right). \tag{60}$$

Compute

$$\{L_j\}_{1\leq j\leq m} = \mathbf{PD}(\sigma, [(Q_i, \theta_i)]_{1\leq i\leq n}). \tag{61}$$

If $m = 0$, return $R' = R$. Otherwise compute a series of MDGs

$$P_{j+1} = \mathbf{PbyS'}(P_j, L_j), \tag{62}$$

starting with $P_1 = R$ and ending as soon as $P_{j+1} = \mathbf{PbyS'}(P_j, L_j) = \mathbf{F}$, or else ending with P_{m+1}. Return as R' the last MDG in the series.

Auxiliary algorithm PD ("push down")

$S = \mathbf{PD}(\sigma, [(Q_i, \theta_i)]_{1\leq i\leq n})$

- Arguments: σ and $[(Q_i, \theta_i)]_{1\leq i\leq n}$ are as before, with $n \geq 0$.
- Result: S is a set (implemented, say, as a list without duplicates) whose elements are sequences of the form $[(R_j, \theta'_j)]_{1\leq j\leq m}$, $m \geq 0$, such that
  - Each R_j is an MDG of type $V \rightsquigarrow Y$, whose top symbol comes after the top symbol of σ
  - Each θ'_j is a substitution mapping variables in V to concretely-reduced terms constructed from variables in V
  - $\models \left(\sigma \wedge \left(\bigwedge_{1\leq j\leq m}(R_j * \theta'_j)\right)\right) \Rightarrow \left(\bigwedge_{1\leq i\leq n}(Q_i * \theta_i)\right)$.

Given two sets S and S' whose elements are sorted lists without duplicates, the *cross-union* of S and S', $S \otimes S'$, is the set of sorted lists without duplicates obtained by merging L and L' for every $(L, L') \in S \times S'$.

If $n = 0$ return $S = \{[\,]\}$. Otherwise let T be the tail of $[(Q_i, \theta_i)]_{1\leq i\leq n}$ (i.e. the sequence whose i-th element if (Q_{i-1}, θ_{i-1}), for $1 \leq i \leq n-1$), and consider the following cases:

1. The top symbol of Q_1 comes after the top symbol of σ in custom order. Then recursively compute

$$S_{\text{tail}} = \mathbf{PD}(\sigma, T). \tag{63}$$

   If $m = 0$, i.e. if $S_{\text{tail}} = \emptyset$, return $S = \emptyset$. Otherwise return

$$S = \{[Q_1, \theta_1]\} \otimes S_{\text{tail}}. \tag{64}$$

2. The top symbol of Q_1 comes before the top symbol of σ. Then:

   (a) If the root of Q_1 is a full node, let Q'_j, $1 \le j \le m$, be the immediate subgraphs of Q_1. For each j, let θ'_j be the restriction of θ_1 to the set of variables having secondary occurrences in Q'_j. Obtain a sorted list L by merging T and $[(Q'_j, \theta'_j)]_{1 \le j \le m}$. Then compute and return

   $$S = \mathbf{PD}(\sigma, L). \tag{65}$$

   (b) Otherwise return $S = \emptyset$.

3. Q_1 and σ have the same top symbol. Then:

   (a) If the common top symbol of Q_1 and σ is a concrete variable y, then σ must consist of a node labeled by y and an edge labeled by an individual constant a. If Q_1 does not have a root edge labeled a, return $S = \emptyset$. If Q_1 has a root edge labeled a, leading to a subgraph Q', recursively compute

   $$S_{\text{tail}} = \mathbf{PD}(\sigma, T). \tag{66}$$

   If S_{tail} is empty, return $S = \emptyset$. Otherwise S_{tail} will be a singleton, $S_{\text{tail}} = \{L\}$. Let θ' be the restriction of θ to the set of variables having secondary occurrences in Q'. Obtain a list L' from L by inserting the pair (Q', θ') at its proper position, if not already present. Return

   $$S = \{L'\}. \tag{67}$$

   (b) If the common top symbol of Q_1 and σ is an abstract variable y, σ must consist of a node labeled by y and an edge labeled by a term A. Let B_j, $1 \le j \le m$, be the labels of the root edges of Q_1 that match A under θ_1, and θ'_j the resulting substitutions: for each j, $1 \le j \le m$, θ'_j is an extension of θ_1 such that $B_j \cdot \theta'_j$ coincides with A. For each j, let Q'_j be the subgraph of Q_1 that the edge B_j leads to, and let θ''_j be the restriction of θ'_j to the set of variables having secondary occurrences in Q'_j.
   If $m = 0$ (i.e. if no matches are found), return $S = \emptyset$. Otherwise recursively compute

   $$S_{\text{tail}} = \mathbf{PD}(\sigma, T). \tag{68}$$

If $S_{\text{tail}} = \emptyset$, return $S = \emptyset$. Otherwise compute and return

$$S = \{[(Q'_j, \theta''_j)]\}_{1 \leq j \leq m} \otimes S_{\text{tail}}. \tag{69}$$

(c) If the common top symbol of Q_1 and σ is a cross-operator f, then (i) σ consists of a series of node-edge pairs $(A_j, a_j)_{1 \leq j \leq m}$, where each A_j is a cross-term with cross-operator f and each a_j is an individual constant, and (ii) Q_1 has a root node labeled by a cross-term B with cross-operator f, and root edges b_k, $1 \leq k \leq l$, labeled by individual constants, leading to subgraphs Q'_k.

For each j, $1 \leq j \leq m$, try to find an extension θ'_j of θ_1 such that $B \cdot \theta'_j$ coincides with A_j, and a k_j such that a_j and b_{k_j} are the same individual constant. If successful, let θ''_j be the restriction of θ'_j to the set of variables having secondary occurrences in Q'_{k_j} and recursively compute

$$S'_j = \mathbf{PD}(\sigma, [(Q'_{k_j}, \theta''_j)]). \tag{70}$$

Otherwise let $S'_j = \emptyset$.

Then, if the root of Q_1 is a full node, let θ'''_k, $1 \leq k \leq l$, be the restriction of θ_1 to the set of variables having secondary occurrences in Q'_k, and recursively compute

$$S'_0 = \mathbf{PD}(\sigma, [(Q'_k, \theta'''_k)]_{1 \leq k \leq l}). \tag{71}$$

Otherwise let $S'_0 = \emptyset$.

Having obtained S'_0 and the S'_j, $1 \leq j \leq m$, compute

$$S_{\text{head}} = \bigcup_{0 \leq j \leq m} S'_m. \tag{72}$$

If $S_{\text{head}} = \emptyset$, return $S = \emptyset$. Otherwise recursively compute

$$S_{\text{tail}} = \mathbf{PD}(\sigma, T) \tag{73}$$

and return

$$S = S_{\text{tail}} \otimes S_{\text{head}}. \tag{74}$$

A.5 Proof of soundness of PbyS

It is straightforward to show that the auxiliary algorithm **PD** is sound and terminating, i.e. that an invocation

$$\mathbf{PD}(\sigma, [(Q_i, \theta_i)]_{1\leq i\leq n}) \tag{75}$$

of the algorithm with appropriate arguments σ and $[(Q_i, \theta_i)]_{1\leq i\leq n}$ terminates and returns a set S satisfying the conditions specified above on the result.

It is also easy to see that the soundness and termination of **PbyS** follow from the soundness and termination of **PbyS'**. (Note that $(\exists V)Q$ is the same as $Q * \theta$ up to renaming of bound variables.) To establish the soundness and termination of **PbyS'**, consider an invocation

$$\mathbf{PbyS'}(P, [(Q_i, \theta_i)]_{1\leq i\leq n}) \tag{76}$$

of the algorithm where the arguments P and $[Q_i]_{1\leq i\leq n}$ are as specified in the description of the algorithm. We are going to prove by induction on the structure of the first argument P that for every appropriate second argument $[(Q_i, \theta_i)]_{1\leq i\leq n}$ the invocation terminates and produces a result P' that satisfies (57).

If $P = \mathbf{T}$ and $n = 0$, **PbyS'** immediately returns $P' = \mathbf{F}$. In this case (57) reduces to

$$\models \mathbf{T} \vee \mathbf{T} \Leftrightarrow \mathbf{F} \vee \mathbf{T} \tag{77}$$

which obviously holds. If $P = \mathbf{T}$ with $n \neq 0$, **PbyS'** returns $P' = \mathbf{T}$ and (57) becomes

$$\models \mathbf{T} \vee \left(\bigwedge_{1\leq i\leq n} (Q_i * \theta_i) \right) \Leftrightarrow \mathbf{T} \vee \left(\bigwedge_{1\leq i\leq n} (Q_i * \theta_i) \right) \tag{78}$$

which again holds. If $P \neq \mathbf{T}$ then, since $P \neq \mathbf{F}$ by hypothesis, the root N of P is not a leaf node. For each segment σ issuing from N and leading to a subgraph R, the algorithm calls **PbyS"** to compute R',

$$R' = \mathbf{PbyS"}(\sigma, R, [(Q_i, \theta_i)]_{1\leq i\leq n}). \tag{79}$$

PbyS" in turn calls **PD** to compute a set of lists

$$\{L_j\}_{1\leq j\leq m} = \mathbf{PD}(\sigma, [(Q_i, \theta_i)]_{1\leq i\leq n}). \tag{80}$$

($m \geq 0$). **PD** terminates, and each L_j is a list $[(R_{jk}, \eta_{jk})]_{1 \leq k \leq l_j}$. Using L_j as a shorthand for

$$\bigwedge_{1 \leq k \leq l_j} R_{jk} * \eta_{jk} \tag{81}$$

we have for each j, $1 \leq j \leq m$,

$$\models \sigma \wedge L_j \Rightarrow \left(\bigwedge_{1 \leq i \leq n} (Q_i * \theta_i) \right). \tag{82}$$

If $m = 0$, **PbyS"** returns $R' = R$, which obviously satisfies (57). Otherwise it computes

$$P_{j+1} = \textbf{PbyS'}(P_j, L_j), \tag{83}$$

starting from $j = 1$, until $j = m$ or **PbyS'** returns **F**; the last value obtained from **PbyS'** is returned as R'. By induction hypothesis each invocation of **PbyS'** terminates and

$$\models P_j \vee L_j \Leftrightarrow P_{j+1} \vee L_j. \tag{84}$$

Hence

$$\models \sigma \wedge (P_j \vee L_j) \Leftrightarrow \sigma \wedge (P_{j+1} \vee L_j) \tag{85}$$

and

$$\models (\sigma \wedge P_j) \vee (\sigma \wedge L_j) \Leftrightarrow (\sigma \wedge P_{j+1}) \vee (\sigma \wedge L_j). \tag{86}$$

Then, by (82),

$$\models (\sigma \wedge P_j) \vee \left(\bigwedge_{1 \leq i \leq n} (Q_i * \theta_i) \right) \Leftrightarrow (\sigma \wedge P_{j+1}) \vee \left(\bigwedge_{1 \leq i \leq n} (Q_i * \theta_i) \right) \tag{87}$$

and, since this holds at every step j of the stepwise computation of R' from R,

$$\models (\sigma \wedge R) \vee \left(\bigwedge_{1 \leq i \leq n} (Q_i * \theta_i) \right) \Leftrightarrow (\sigma \wedge R') \vee \left(\bigwedge_{1 \leq i \leq n} (Q_i * \theta_i) \right). \tag{88}$$

Having computed R' for each σ and R, **PbyS'** concludes by replacing each R with R', eliminating **F**, and removing redundant nodes. Clearly, P is logically equivalent to the disjunction of the formulas $\sigma \wedge R$, and P' to the disjunction of the formulas $\sigma \wedge R'$. Thus, since we have established (88) for every R and corresponding R',

$$\models P \vee \left(\bigwedge_{1 \leq i \leq n} (Q_i * \theta_i) \right) \Leftrightarrow P' \vee \left(\bigwedge_{1 \leq i \leq n} (Q_i * \theta_i) \right). \tag{89}$$

We have thus proved the termination and soundness of **PbyS'**.