A High Level Reachability Analysis using Multiway Decision Graph in the HOL Theorem Prover

Sa'ed Abed, Otmane Ait Mohamed and Ghiath Al Sammane

Department of Electrical and Computer Engineering,
Concordia University, Montreal, Canada
{s_abed,ait,sammane}@ece.concordia.ca

Abstract. In this paper, we provide all the necessary infrastructure to define a high level states exploration approach within the HOL theorem prover. While related work has tackled the same problem by representing primitive BDD operations as inference rules added to the core of the theorem prover, we have based our approach on the Multiway Decision Graphs (MDGs). We define canonic MDGs as well-formed directed formulae in HOL. Then, we formalize the basic MDG operations following a deep embedding approach and we derive the correctness proof for each operation. Finally, a high level reachability analysis is implemented as a tactic that uses our MDG theory within HOL.

1 Introduction

Model checking and deductive theorem proving are the two main formal verification approaches of digital systems. It is accepted that both approaches have complementary strengths and weaknesses. Model checking algorithms can automatically decide if a temporal property holds for a finite-state system. They can produce a counterexample when the property does not hold, which can be very important for the reparation of the corresponding error in the implementation under verification or in the specification itself. However, model checking suffers from the states explosion problem when dealing with complex systems. In deductive reasoning, the correctness of a design is formulated as a theorem in a mathematical logic and the proof of the theorem is checked using a general-purpose theorem-prover. This approach can handle complex systems but requires skilled manual guidance for verification and human insight for debugging. Unfortunately, if the property fails to hold, deductive methods do not give counterexample.

Indeed, the combination of the two approaches, states exploration and deductive reasoning promises to overcome the limitations and to enhance the capabilities of each. Our research is directed toward this goal. In this paper, we provide all the necessary infrastructure (data structure + algorithms) to define a high level state exploration in the HOL theorem prover. While related work has tackled the same problem by representing primitive Binary Decision Diagrams

(BDD) operations [4] as inference rules added to the core of the theorem prover [7], we have based our approach on the Multiway Decision Graphs (MDGs) [5]. MDG generalizes ROBDD to represent and manipulate a subset of first-order logic formulae, which is more suitable for defining model checking inside a theorem prover. With MDGs, a data value is represented by a single variable of an abstract type, and operations on data are represented in terms of uninterpreted functions. Considering MDG instead of BDD will raise the abstraction level of what can be verified using a state exploration within a theorem prover. Therefore, an MDG structure in HOL allows better proof automation for larger datapaths systems.

The paper is organized as follows: First, we define the MDG structure inside the HOL system in order to be able to construct and manipulate MDG as formulae in HOL. This step implies a formal logic representation for the MDG or what we call: *The MDG Syntax*. It is based on the Directed Formulae DF: an alternative vision for the MDG in terms of logic and set theory [2]. Subsequently, all the basic operations are built on the top of *The MDG Syntax*. Then, the definitions of the MDG operations, following a deep embedding approach in HOL, are associated naturally with a proof of their correctness. Finally, we define an MDG based reachability analysis in HOL as a tactic that uses the MDG theory.

2 Related Work

The closest work, in approach, to our own are those of Gordon [7, 6] and later Amjad [3]. Gordon integrated the verification system BuDDy (BDD package implemented in C) into HOL. The aim of using BuDDy is to get near the performance of C-based model checker, whilst remaining fully expansive, though with a radically extended set of inference rules [6, 7].

In [8], Harrison implemented BDDs inside HOL without making use of an external oracle. The BDD algorithms were used by a tautology-checker. However, the performance was a thousand times slower than a BDD engine implemented in C. Harrison mentioned that by re-implementing some of HOL's primitive rules, the performance could be improved by around ten times.

Amjad [3] demonstrated how BDD based symbolic model checking algorithms for the propositional $\mu - calculus$ (L_μ) can be embedded in the HOL theorem prover. This approach allows results returned from the model checker to be treated as theorems in HOL. The approach still leaves results reliant on the soundness of the underlying BDD tools. Therefore, the security of the theorem prover is compromised to the extent that the BDD engine or the BDD inference rules may be unsound. Our work focusses more on how one can raise the level of assurance by embedding and proving formally the correctness of those operators in HOL.

In fact, while BDDs are widely used in state-exploration methods, they can only represent Boolean formulae. Our work deals with the embedding of MDGs rather than BDDs. The work of Mhamdi and Tahar [10] builds on the MDG-HOL [9] project, but uses a tightly integrated system with the MDG primitives written in ML rather than two tools communicating as in MDG-HOL system. The syntax is partially embedded and the conditions for well-formedness must be respected by the user. By contrast, we provide a complete embedding of the MDG syntax and the conditions could be checked automatically in HOL.

Another major difference between the above work and ours is that it implements the related inference rules for BDD operators in the core of HOL as un-trusted code, whereas we implement the MDG operations as trusted code in HOL.

Verification of BDD algorithms has been a subject of active research and many papers offer studies conducted using different proof assistants such as HOL, PVS, Coq and ACL2. These papers tries to extend the prover with a verified BDD package to enhance the BDD performance, while still inside a formal proof system. For example, in [13] the correctness of BDD algorithms using Coq has been proved. The goal is to extract a verified algorithms manipulating BDDs in Caml (the implementation language of Coq).

Our work follows the verification of the Boolean manipulating package, but using MDG instead. The correctness of the model checker is then, defined as a tactic, can be achieved after the proof of these algorithms.

3 Multiway Decision Graphs

MDGs subsume the class of Bryant's (ROBDD) [4] while accommodating abstract data and uninterpreted function symbols. It can be seen as a Directed Acyclic Graph (DAG) with one root, whose leaves are labeled by formulae of the logic True (T)[5]. The internal nodes are labeled by terms, and the edges issuing from an internal nodes v are labeled by terms of the same sort as the label of v. Terms are made out of sorts, constants, variables, and function symbols. Two kinds of sorts are distinguished: concrete and abstract:

– Concrete sort: is equipped with finite enumerations, lists of individual constants. Concrete sorts is used to represent control signals.
– Abstract sort: has no enumeration available. It uses first order terms to represent data signals.

Figure 1 shows two MDG examples G0 and G1. In G0, X is a variable of the concrete sort $[0, 2, 3]$, while in G1, X is a variable of abstract sort; α, β and $f(\theta)$ are abstract terms.

MDGs are canonical representation, which means that an MDG structure has: a fixed node order, no duplicate edges, no redundant nodes, no isomorphic subgraphs, terms concretely reduced that have no concrete subterms other than individual constants, disjoint primary (nodes label) and secondary variables (edges label).

Different approaches have been used to formalize decision diagrams either as terms and formulae or as DAGs. The first is a formal logic representation using data type definitions [7, 3], while the later is a graphical representation using trees and graphs [12, 13].

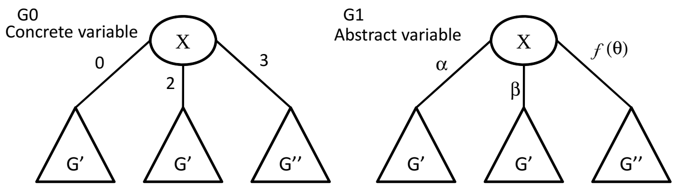

Fig. 1. Example of Multiway Decision Graphs Structure

The choice between the two approaches depends on the formalization objectives. If we want to reason about the correctness of the implementation itself, then we need to define decision diagrams as graphs and do sharing of common sub-trees. Clearly this makes the development and the proofs complex. On the other hand, if we are only interested in a high-level view, for the use of induction, then a logical representation is preferred. This is why, we choose the logical representation in terms of Directed Formulae (DF) to model the MDG syntax in HOL.

3.1 Directed Formulae (DF)

Let $\mathcal{F}$ be a set of function symbol and $\mathcal{V}$ a set of variables. We denote the set of terms freely generated from $\mathcal{F}$ and $\mathcal{V}$ by $\mathcal{T}(\mathcal{F},\mathcal{V})$. The syntax of a Directed Formula is given by the grammar below. The underline is used to differentiate between the concrete and abstract variables.

Sort S	$::= S \mid \underline{S}$
Abstract Sort S	$::= \alpha \mid \beta \mid \gamma \mid \cdots$
Concrete Sort $\underline{S}$	$::= \underline{\alpha} \mid \underline{\beta} \mid \underline{\gamma} \mid \cdots$
Generic Constant C	$::= a \mid b \mid c \mid \cdots$
Concrete Constant $\underline{C}$	$::= \underline{a} \mid \underline{b} \mid \underline{c} \mid \cdots$
Variable $\mathcal{X}$	$::= V \mid \underline{V}$
Abstract Variable V	$::= x \mid y \mid z \mid \cdots$
Concrete Variable $\underline{V}$	$::= \underline{x} \mid \underline{y} \mid \underline{z} \mid \cdots$
Directed formulae DF	$::= Disj \mid \top \mid \bot$
$Disj$	$::= Conj \vee Disj \mid Conj$
$Conj$	$::= Eq \wedge Conj \mid Eq$
Eq	$::= \underline{A} = \underline{C} \quad (A \in \mathcal{T}(\mathcal{F}, V))$
	$\mid \underline{V} = \underline{C}$
	$\mid V = A \quad (A \in \mathcal{T}(\mathcal{F}, \mathcal{X}))$

The vocabulary consists of generic constants, concrete constants (individual), abstract variables, concrete variables and function symbols. DF are always disjunction of conjunctions of equations or $\top$ (truth) or $\bot$ (false). The conjunction *Conj* is defined to be an equation only *Eq* or a conjunction of at least two equations. Atomic formulae are the equations, generated by the clause Eq. The equation can be the equality of concrete term and an individual constant, the equality of a concrete variable and an individual constant, or the equality of an abstract variable and an abstract term.

Given two disjoint sets of variables U and V, a *Directed Formulae* of type $U \rightarrow V$ is a formula in Disjunctive Normal Form (DNF). Just as ROBDD must be *reduced* and *ordered*, DFs must obey a set of well-formedness conditions given in [5] such that:

1. Each disjunct is a conjunction of equations of the form:
   $A = a$, where A is a term of concrete sort α containing no variables other than elements of U, and a is an individual constant in the enumeration of α, or
   $u = a$, where $u \in (U \cup V)$ is a variable of concrete sort α and a is an individual constant in the enumeration of α, or
   $v = A$, where $v \in V$ is a variable of abstract sort α and A is a term of type α containing no variables other than elements of U;
2. In each disjunct, the LHSs of the equations are pairwise distinct; and
3. Every abstract variable $v \in V$ appears as the LHS of an equation $v = A$ in each of the disjuncts. (Note that there need not be an equation $v = a$ for every concrete variable $v \in V$).

DFs are used for two purposes: to represent sets (viz. sets of states as well as sets of input vectors and output vectors) and to represent relations (viz. the transition and output relations).

For example, suppose that $U = \{u1, u2\}$ and $V = \{v1, v2\}$, where $u1$ and $v1$ are variables of concrete sort *bool* with enumeration $\{0, 1\}$ while $u2$ and $v2$ are variables of an abstract sort *wordn*. Also, suppose that f is an abstract function symbol of type *wordn* $\rightarrow$ *wordn* and g is a cross-operator of type *wordn* $\rightarrow$ *bool*. Then, Figure 1 shows the MDG representing this example as well as its corresponding DF formula.

3.2 MDG Operations

We give the definitions of MDG basic operations in terms of DF's [5].

Conjunction Operation: The conjunction operation takes as inputs two DFs P_i, $1 \leq i \leq 2$, of types $U_i \rightarrow V_i$, and produce a DF $R = \textbf{Conj}\ (\{P_i\}_{1 \leq i \leq 2})$ of type $(\bigcup_{1 \leq i \leq 2} U_i) \backslash (\bigcup_{1 \leq i \leq 2} V_i) \rightarrow (\bigcup_{1 \leq i \leq 2} V_i)$ such that:

$$\models R \Leftrightarrow (\bigwedge_{1 \leq i \leq 2} P_i) \qquad (1)$$

Note that for $1 \leq i < j \leq 2$, V_i and V_j must not have any abstract variables in common, otherwise the conjunction cannot be computed.

Relational Product Operation: The relational product performs conjunction and existential quantifying for a two DFs. It is used for image computation.

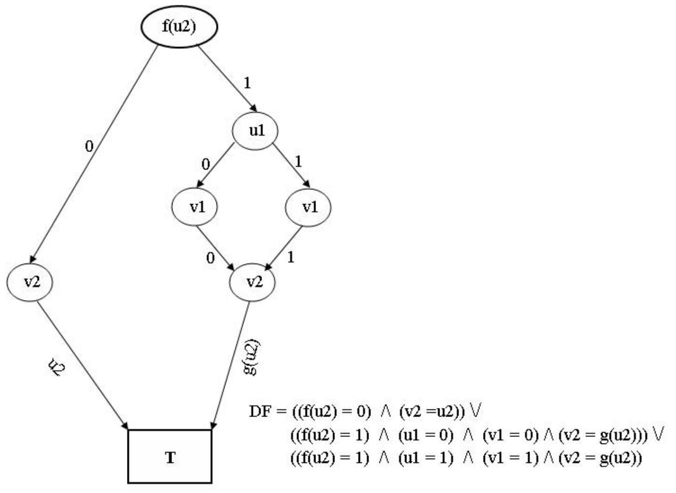
f(u2)
0
1
u1
0
1
v1
v1
0
1
v2
v2
u2
g(u2)
T
DF = ((f(u2) = 0) ∧ (v2 =u2)) ∨
((f(u2) = 1) ∧ (u1 = 0) ∧ (v1 = 0) ∧ (v2 = g(u2))) ∨
((f(u2) = 1) ∧ (u1 = 1) ∧ (v1 = 1) ∧ (v2 = g(u2))

Fig. 2. MDG and its Corresponding DF

Disjunction Operation: The disjunction operation performs disjunction for two DFs having the same set of abstract primary variables.

Pruning By Subsumption: The pruning by subsumption takes as inputs two DFs P and Q of types $U \rightarrow V_1$ and $U \rightarrow V_2$ respectively, where U contains only abstract variables that do not participate in the symbol ordering, and produces a DF $R =$ **PbyS** (P, Q) of type $U \rightarrow V_1$ derivable from P by *pruning* (i.e. by removing some of disjoints) such that:

$$\models R \vee (\exists E)Q \Leftrightarrow P \vee (\exists E)Q \qquad (2)$$

The disjuncts that are removed from P are *subsumed* by Q, hence the name of the algorithm.

Since R is derivable from P by pruning, after the formulae represented by R and P have been converted to DNF, the disjuncts in the DNF of R are a subset of those in the DNF of P. Hence $\models R \Rightarrow P$. And, from (2), it follows tautologically that $\models P \wedge \neg(\exists E)Q \Rightarrow R$. Thus we have

$$\models (P \wedge \neg(\exists E)Q \Rightarrow R) \wedge (R \Rightarrow P)$$

We can then view R as approximating the logical difference of P and $(\exists E)Q$.

4 The MDG Syntax

4.1 DF in HOL

We use HOL recursive datatype to define the MDG syntax. an MDG sort is either concrete or abstract. This is embedded using two constructors called `Abst_Sort` and `Conc_Sort`. The `Abst_Sort` takes as argument an abstract sort name of type *alpha* and the `Conc_Sort` takes a concrete sort name and its enumeration of type *string* as an input argument. This is declared in HOL as follows:

```
Sort ::= Abst_Sort of 'alpha | Conc_Sort of string →  string list
```

To determine whether the sort is concrete or abstract, we define predicates over the constructor called `Is_Abst_Sort` and `Is_Conc_Sort`.

In the same way, constants and variables are either of concrete or abstract sort. Individual constant can have multiple sorts depending on the enumeration of the sort, while abstract generic constant is identified by its name and its abstract sort. A variable (abstract or concrete) is identified by its name and sort. For example, abstract variable is realized by defining a new HOL type as shown below:

```
Abst_Var ::= Abst_Var of string →  'alpha Sort
```

Functions can be either abstract or cross-operators. Cross-functions are those that have at least one abstract argument:

```
Cross_Fun ::= Cross_Fun of string →  'alpha Var list →  'alpha Sort
```

We have defined a datatype `D_F`. The DF can be True or False or a disjunction of conjunction of equations. Then we define the type definition of a directed formula:

```
D_F ::= DF1 of 'alpha DF | TRUE | FALSE
DF  ::= DISJ of 'alpha MDG_Conj →  DF | CONJ1 of 'alpha MDG_Conj
MDG_Conj ::= Eqn of 'alpha Eqn | CONJ of 'alpha Eqn →  MDG_Conj
Eqn      ::= EQUAL1 of 'alpha Conc_Vari →  'alpha Ind_Cons
           | EQUAL2 of 'alpha Abst_Var →  'alpha Abst_Fun
           | EQUAL3 of 'alpha Cross_Fun →  ('alpha Abst_Var) list →
                                            'alpha Ind_Cons
           | EQUAL4 of 'alpha Abst_Var →  'alpha Abst_Var
           | EQUAL5 of 'alpha Abst_Var →  'alpha Gen_Cons
```

`DF1`, `DISJ`, `CONJ1`, `Eqn`, `CONJ` are distinct constructors and the constructors `EQUAL1, EQUAL2, EQUAL3, EQUAL4, EQUAL5` are used to define an atomic equation. The type definition package returns a theorem which characterizes the type `D_F` and allows reasoning about this type. Note that the type is polymorphic in a sense that the variable could be represented by a string or an integer number or any user defined type; in our case we have used the string type.

4.2 Well-formedness Conditions

Since the DF is represented as a list of equations. The embedding of the well-formedness conditions can be defined straightforward by:

– The first condition is satisfied by construction following the `Eqn` type definition.
– The second condition is embedded as:

```
⊢def  (Condition2 [] = T) ∧
       (Condition2 (hd::tl) = ALL_DISTINCT hd ∧ Condition2 tl)
```

– The embedding of the third condition requires more work and needs an auxiliary function as shown below:

```
⊢def  (Condition3 (hd1::tl1) [] = T) ∧
       (Condition3 [] (hd2::tl2) = T) ∧
       (Condition3 (hd1::tl1) (hd2::tl2) =
           Condition_3 hd1 (hd2::tl2) ∧  Condition3 tl1 (hd2::tl2))
⊢def  (Condition_3 hd1 []=T) ∧
       (Condition_3 hd1 (hd2::tl2)=IS_EL hd1 hd2 ∧ Condition_3 hd1 tl2)
```

Finally, the predicate `Is_Well_Formed_DF` is defined as:

```
⊢def  ∀df. Is_Well_Formed_DF df =
            Condition2 (STRIP_DF df) ∧
            Condition3 (FLAT(STRIP_ABS_DF df)) (STRIP_DF df))
```

where `STRIP_ABS_DF` function extracts the abstract variables of a DF and `STRIP_DF` extracts the LHS variables of each disjuncts of a DF. We have implemented a HOL tactic to automatize the checking of well-formedness conditions [11]. The motivation for the well-formedness conditions is very important since the inputs for the operations must be well-formed as well as the result.

5 Embedding of the MDG Operations

In fact, HOL provides predefined logical operations that perform conjunction and disjunction of formulae. However, if the inputs of these operations are well-formed DF, outputs will not be necessary well-formed DF. Also, as the DF represent a canonical graph, the variables order must be preserved to satisfy the well-formedness conditions, which is not satisfied when applying HOL operations. Our embedding, is built to answer specifically these concerns. In this Section, we provide a formal embedding of MDG basic operations as well as the proof of their correctness.

However, the proof strategy consists of feeding the same inputs to the logical HOL predefined operations and to the embedded MDG operations. As the output of the embedded MDG operation is well-formed DF, it needs a refinement step to obtain formulae. These formulae will be compared with the output formulae of logical HOL operation. We check the equivalence of both and prove it as a theorem using structural induction and rewriting rules. We describe the conjunction and PbyS operations in detail. The PbyS operation is different and represents the core of the reachability analysis algorithm. The disjunction and RelP operations are similar to the conjunction [1]. The complete source code for the embedding is available in [11].

5.1 The Conjunction Operation

The method for computing the conjunction of two DFs is applicable when the sets of primary variables of the two DFs are disjoint. The conjunction operator accepts two sets of DFs (df1 and df2) and the order list L of the node label. The detailed algorithm is given in Algorithm 1.

Algorithm 1 CONJ_ALG (df1, df2, L)

```
1: if (df1 or df2 = terminal DF) then
2:    return result;
3: else
4:    for (each disjunct ∈ df1) do
5:       DF_CONJUNCTION (disj1_df1,df2,L) recursively
6:       for (each disjunct ∈ df2) do
7:          HD_SUBST (HD_DISJUNCT (disjt1_df1,disjt1_df2,L)) recursively
8:       end for
9:       append the result of the HD_DISJUNCT;
10:   end for
11:   append the result of the DF_CONJUNCTION;
12: end if
```

Algorithm 1 starts with two well formed DFs and an order list L. The resulted DF is constructed recursively and ended when a terminal DF (true or false) is reached (lines 1 and 2). Lines 4 to 11 recursively, applies the conjunction between df1 and df2 (`DF_CONJ` function). The `DF_CONJUNCTION` function determines the conjunction of the first disjunct of df1(disj1_df1) and df2 as shown in line 5. More details in the embedding can be found in [1].

The `HD_DISJUNCT` function determines the conjunction between the first disjunct of both DFs (lines 6 to 8). Then, we apply the substitution to be sure that the result is well formed DF using the `HD_SUBST` function. The substitution is carried out by taking the disjunct and check the LHS of each equation (primary variable) does not appear in any equations in the RHS (secondary variable) of the same disjunct. If it appears then we apply substitution by replacing its RHS by the other RHS to respect the well formedness conditions. Line 9 recursively append the result and move to the second disjunct of df1. In line 11, the `DF_CONJUNCTION` function recursively performs the conjunction of the second disjunct of df1 with df2 and append it to the result.

Finally, the conjunction operation is embedded in HOL as:

```
⊢def  ∀df1 df2 L. CONJ_ALG df1 df2 L =
 (if df1 = TRUE then STRIP_DF_list df2
  else if df2 = TRUE then STRIP_DF_list df1
  else if df1 = FALSE then STRIP_DF_list df1
  else if df2 = FALSE then STRIP_DF_list df2
  else TAKE_HD DF_CONJ (STRIP_DF_list df1) (STRIP_DF_list df2)
              (union (STRIP_Fun df1) (STRIP_Fun df2)) L)
```

We prove the correctness of the conjunction operation as shown in Theorem 1. The detailed proof can be found in [1].

Theorem 1. *Conjunction Correctness*
Let df1 and df2 be well formed DF. Let L be an order list that is equal to the union of their order lists. Then, the MDG conjunction of df1 and df2 (`CONJ_ALG`), and HOL logical conjunction of df1 and df2 (`CONJ_LOGIC`: mapping a list to disjunction of conjunction of equations), are equivalent.

```
Conjunction Correctness ⊢  ∀df1 df2. ∃L. Is_Well_Formed_DF df1 ∧
   Is_Well_Formed_DF df2 ∧ (ORDER_LIST df1 df2 = L) ⟹
   (CONJ_LOGIC df1 df2 = DISJ_LIST (CONJ_ALG df1 df2 L))
```

Proof. The goal is to prove the equivalence of MDG conjunction and HOL logical conjunction for these DF. The proof uses structural induction on df1 and df2 and rewriting rules. □

where the `CONJ_LOGIC` function represents the HOL logical conjunction of df1 and df2 (mapping a list to disjunction of conjunction of equations).

5.2 Pruning by Subsumption (PbyS) Operation

The pruning by subsumption operation is used to approximate the difference of sets represented by DFs. Informally, it removes all the paths of a DF P from another DF Q. The constraints for PbyS requires as inputs two well-formed DFs of types $U \rightarrow V_1$ and $U \rightarrow V_2$, respectively. Also, an order list L that represents the union of the two DFs order lists (pre-conditions) is needed. The constraint related to the execution is: the list of variables U should contain only abstract variables that do not participate in L. The result of the algorithm must be a well-formed DF that represents the pruning by subsumption of df1 and df2, and of the same type as df1 $U \rightarrow V_1$ (post-condition).

Algorithm 2 starts with two well formed DFs and order list L. The resulted DF is constructed recursively and ended when a terminal DF (true or false) is reached (lines 1 and 2). Line 3 checks the equality of both RHS abstract variables of df1 and df2. If they are equal, then the algorithm checks if those abstract variables are not included in the order list L using the function `IS_ABS_IN_ORDER` (line 4). Otherwise, it returns an empty list (line 10). If the condition is satisfied, then the algorithm determines the pruning by subsumption of the two DFs by calling `DF_PbyS` function (line 5). Otherwise, the algorithm returns an empty list (line 7).

Finally, the pruning by subsumption operation is:

Algorithm 2 PbyS_ALG (df1, df2, L)

```
 1: if (df1 or df2 = terminal DF) then
 2:    return result;
 3: else if (STRIP_ABS_RHS_DF df1 = STRIP_ABS_RHS_DF df2) then
 4:    if (STRIP_ABS_RHS_DF df1 ∉ L) then
 5:       call DF_PbyS(df1, df2);
 6:    else
 7:       return empty list;
 8:    end if
 9: else
10:    return empty list;
11: end if
```

```
⊢def  ∀df1 df2 L. PbyS_ALG df1 df2 L =
        if (df1 = TRUE) then [[["FALSE"]]]
        else if (df2 = TRUE) then [[["FALSE"]]]
        else if (df1 = FALSE) then [[["FALSE"]]]
        else if (df2 = FALSE) then (STRIP_DF_list df1)
        else if (IS_ABS_IN_ORDER(FLAT(STRIP_ABS_RHS_DF df2))L=[]) then
              if (IS_ABS_IN_ORDER(FLAT(STRIP_ABS_RHS_DF df1))L=[]) then
                DF_PbyS (STRIP_DF_list df1) (STRIP_DF_list df2)
                        (union (STRIP_Fun df1) (STRIP_Fun df2))
                        (HD_l_abs(STRIP_DF_l_abs_list df1))
                        (HD_l_abs(STRIP_DF_l_abs_list df2)) L
              else  []
        else  []
```

We show here the correctness proof of PbyS operation in Theorem 2.

Theorem 2. *Pruning by Subsumption Correctness*
Let df1 and df2 be well formed DF. Let L be an order list that is equal to the union of their order lists. Then, the MDG disjunction of PbyS_ALG(df1,df2,L) and ((EXIST_QUANT U) df2), is equivalent to the HOL disjunction of df1 and ((EXISTS_LIST U) df2):

```
Pruning by Subsumption Correctness ⊢  ∀df1 df2. ∃L1. ∃L2.
  Is_Well_Formed_DF df1 ∧
  Is_Well_Formed_DF df2 ∧ (ORDER_LIST df1 df2 = L1)  ⟹
  ((DISJ_LIST (STRIP_DF_list df1) ∨
    DISJ_LIST(EXISTS_LIST(STRIP_DF_list df2) L2)) =
   (DISJ_LIST (PbyS_ALG df1 df2 L1) ∨
    DISJ_LIST(EXIST_QUANT(STRIP_DF_list df2) L2)))
```

Proof. The proof uses structural induction on df1 and df2 and rewriting rules. □

This algorithm is used to check whether a set of states is a subset of another set of states. Let df1, df2 be two DFs of type $U \to V$, then we say that df1 and df2 are *equivalent* DFs if `PbyS(df1,df2,L) = PbyS(df2,df1,L)`:

```
Equivalence ⊢  ∀df1 df2. ∃L. Is_Well_Formed_DF df1 ∧
   Is_Well_Formed_DF df2 ∧ (ORDER_LIST df1 df2 = L) ∧
   (DISJ_LIST(PbyS_ALG df1 df2 L) = DISJ_LIST(PbyS_ALG df2 df1 L))
       ⟹  (df1 = df2)
```

In fact, the conjunction operation has consumed most of the proof preparation effort. Most of the definitions and proofs are reused by the other operations, especially the relational product operation. The embedding of MDG syntax and the verification of MDG operations sums up to 14000 lines of HOL codes. The complexity of the proof is related mainly to the MDG structure, and the recursive definitions of MDG operations.

6 The Reachability Analysis

We show here, the steps to compute the reachability analysis [5] of an abstract state machine using our MDG operations. The important difference is that we are using our embedded DF operators in a high level. At this stage, the proof expert reasons directly in terms of DF, the internal list representation that we have used in the proof of operations is completely encapsulated.

The non-termination problem Due to the abstract representation and the uninterpreted function symbols, the reachability analysis algorithm may not terminate [5]. Several practical solutions have been proposed to solve the non termination problem. The authors in [2] related the problem to the nature of the analyzed circuit. Furthermore, they have characterized some mathematical criteria that leads explicitly to the non termination of particular classes of circuits. Thus, we follow a practical consideration for the MDG reachability. Instead of embedding the theory and the algorithms in general, we rather embed the reachability computation of a particular circuit (DF). Our tactic can be applied to any circuit, but cannot prove the general MDG reachability correctness. We illustrate our technique using the MIN-MAX example.

The MIN-MAX Illustrative Example

We consider the MIN-MAX circuit described in [5]. The MIN-MAX state machine shown in Figure 3 has two input variables $X = \{r; x\}$ and three state variables $Y = \{c; m; M\}$, where r and c are of the Boolean sort B, a concrete sort with enumeration $\{0; 1\}$, and x, m, and M are of an abstract sort s. The outputs coincide with the state variables, i.e. all the state variables are observable and there are no additional output variables. The machine stores in m and M, respectively, the smallest and the greatest values presented at the input x since the last reset ($r = 1$). The min and max symbols are uninterpreted generic constants of sort s. The DFs of the individual transition relations, for a particular custom symbol order, are shown below:

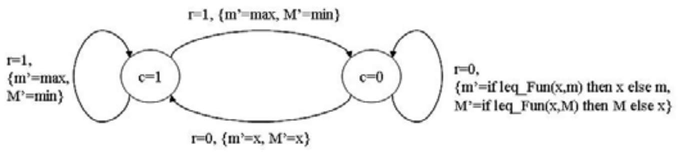
r=1, {m'=max, M'=min}
r=1, {m'=max, M'=min}
c=1
c=0
r=0, {m'=if leq_Fun(x,m) then x else m, M'=if leq_Fun(x,M) then M else x}
r=0, {m'=x, M'=x}

Fig. 3. MIN-MAX State Machine

$$
\begin{aligned}
Tr\_c &= [((r=0)\wedge(n\_c=0))\bigvee((r=1)\wedge(n\_c=1))] \\
Tr\_m &= [((r=0)\wedge(c=0)\wedge(n\_m=m)\wedge(leq\_Fun(x,m)=0))\bigvee \\
&\quad ((r=0)\wedge(c=0)\wedge(n\_m=x)\wedge(leq\_Fun(x,m)=1))\bigvee \\
&\quad ((r=0)\wedge(c=1)\wedge(n\_m=x))\bigvee((r=1)\wedge(n\_m=max))] \\
Tr\_M &= [((r=0)\wedge(c=0)\wedge(n\_M=x)\wedge(leq\_Fun(x,M)=0))\bigvee \\
&\quad ((r=0)\wedge(c=0)\wedge(n\_M=M)\wedge(leq\_Fun(x,M)=1))\bigvee \\
&\quad ((r=0)\wedge(c=1)\wedge(n\_M=x))\bigvee((r=1)\wedge(n\_M=min))]
\end{aligned}
$$

The DF of the system transition relation Tr is the conjunction of these individual transition relations. Firstly, we illustrate how the well-formedness conditions are checked. We give partially the definitions for the corresponding MDG syntax:

```
⊢def  bool = Conc_Sort "bool" ["0";"1"]
⊢def  wordn = Abst_Sort "wordn"
⊢def  oone = Ind_Cons "1" bool
⊢def  eq2 = EQUAL1 ^r ^oone
⊢def  mdg1  = CONJ ^eq2 (CONJ ^eq4 (CONJ ^eq11 (Eqn ^eq16)))
```

Then, the directed formula Tr is defined as:

```
⊢def  Tr = DF1 (DISJ ^mdg1 (DISJ ^mdg2 (DISJ ^mdg3
                 (DISJ ^mdg4 (DISJ ^mdg5 (CONJ1 ^mdg6))))))
```

Applying the predicate `Is_Well_Formed_DF` (conversion tactic) on the above directed formula will result true; in the form of theorem.

```
⊢  Is_Well_Formed_DF Tr
```

Secondly, we define one reachability computational step: `Reach_Step`. It takes as inputs: the set of input variables `I`, the set of initial states `Q`, the transition relation `Tr`, the state variables to be renamed `Ren` and the order list `L`.

`Reach_Step` computes the next reachable state by applying successively `Union_Step` which calls `Next_State` and `Frontier_Step`. The `Next_State` computes the set of next states reached from a set of given states with respect to the transition relation of the MIN-MAX. The result is obtained using the embedded DF relational product operator `RelP`. The `Frontier_Step` is used to check if all the states reachable by the machine are already visited. Then, the

Union_Step merges the output of Frontier_Step with the set of states reached previously using the PbyS embedded in Section 5.2 and disjunction operators [1], respectively.

The function RA_n n representing the set of states reachable in n or fewer steps is then defined recursively by

```
⊢def  (RA_n (0) I I_F Q Q_F Tr Tr_F Tr_A E Ren L R R_F R_A = R) ∧
     (RA_n (SUC n) I I_F Q Q_F Tr Tr_F Tr_A E Ren L R R_F R_A =
        (Reach_Step I I_F
        (Frontier_Step I I_F Q Q_F Tr Tr_F Tr_A E Ren L
        (RA_n n I I_F Q Q_F Tr Tr_F Tr_A E Ren L R R_F R_A) R_F R_A)
        Q_F Tr Tr_F Tr_A E Ren L
        (RA_n n I I_F Q Q_F Tr Tr_F Tr_A E Ren L R R_F R_A) R_F R_A ))
```

the variables (v=I I_F Q Q_F Tr Tr_F Tr_A In Ren L R R_F R_A) are extracted from the initialization step. Then, to compute the set of reachable states we need to compute RA_n 0 v, RA_n 1 v, RA_n 2 v etc. Note that the computation of RA_n (n+1) v needs the computation of RA_n n v.

Then, we define the MDG reachability analysis Re_An by calling RA_n:

```
⊢def  (Re_An n I Q Tr E Ren L =
      RA_n n (STRIP_DF_list I) (STRIP_Fun I)
             (STRIP_DF_list Q) (STRIP_Fun Q)
             (STRIP_DF_list Tr)
             (STRIP_Fun Tr) (HD_l_abs(STRIP_DF_l_abs_list Tr)) E Ren L
             (rep_list(STRIP_DF_list Q))
             (STRIP_Fun Q) (HD_l_abs(STRIP_DF_l_abs_list Q)) )
```

Re_An terminates if we reach a fixpoint characterized by an empty frontier set. That for some particular n, say n=n0, eventually:

```
RA_n n I I_F Q Q_F Tr Tr_F Tr_A In Ren L R R_F R_A =
RA_n (n+1) I I_F Q Q_F Tr Tr_F Tr_A In Ren L R R_F R_A
```

This condition is tested at each stage and raise an exception (fixpoint not yet reached) or return a success (the set of reachable states).

The set of initial states is described by the DF $Q0$:

$$Q0 = [((c = 1) \wedge (m = max) \wedge (M = min))]$$

Also the initial reachable states is $R0 = Q0$. Then, for the first Reach_Step the reachable states are:

$$R1 = [((c = 0) \wedge (m = x1) \wedge (M = x1)) \bigvee ((c = 1) \wedge (m = max) \wedge (M = min))]$$

we achieve a fixpoint after three Reach_Step calls and the reachable states at the third iteration (for this example)$R2$:

$$\begin{aligned} R2 = [&((c=0) \wedge (m=x1) \wedge (M=x2)) \wedge (leq\_Fun(x1,x2)=0) \bigvee \\ &((c=0) \wedge (m=x2) \wedge (M=x1)) \wedge (leq\_Fun(x2,x1)=1) \bigvee \\ &((c=1) \wedge (m=max) \wedge (M=min))] \end{aligned}$$

Finally, we prove the following fixpoint theorem by instantiating the parameters of MIN-MAX:

```
Fixpoint ⊢  ∃n0. ∀n. (n>n0) ⟹
    (Re_An (SUC n) ^I ^Q0 ^Tr ^E ^Ren ^L ^Q0 =
     Re_An n ^I ^Q0 ^Tr ^E ^Ren ^L ^Q0)
```

The Generalized Tactic The previous reachability analysis can be generalized as a tactic in order to be applied on other circuits. What will change is only the DF and the set of initial states, if we consider the order list is given. Then our tactic (conversion) in its generalized form can be applied to any DF of a circuit. However, the proof of the reachability fixpoint depends on the structure of the circuit and cannot be considered a general solution to the non-termination problem. The tactic encapsulates the following steps:

1. Formalize the circuit in terms of DF.
2. Check for WF conditions.
3. Formalize `Reach_Step`.
4. Formalize `RA_n`.
5. Prove fixpoint of `Re_An`.

The advantage is that we compute the reachable states for only one iteration and then relying on the induction power in HOL we prove reaching a fixpoint. However, this fixpoint may not exist for some particular circuits. Furthermore, finding the induction scheme is not always a trivial step. If the execution path of the circuit is explicitly inductive like for example a circuit that implements the factorial. Then, the inductive variables are identified easily. For most cases, some knowledge of the circuit is needed as the induction is not explicitly identified as shown in the MIN-MAX example.

7 Conclusion and Future Work

MDGs have been proposed to extend BDD in the representation of the relations as well as sets of states, in terms of abstract sorts to denote data values and uninterpreted function symbols to denote data operations. We have MDG as formulae in high order logic using the Directed Formula notations. The formalization of the basic MDG operations is built on the top of our MDG syntax. Internally, we have used a list representation for the DF that is more efficient for the embedding and for the correctness proof. The reachability analysis is performed using our platform: we have shown how a fixpoint computation can be used to prove the existence of such a fixpoint, depending on the DF circuit structure. Here, the proof is planned over the DF as formulae instead of list representation which raises the abstraction during the proof of reachability.

This work is an important step, to define a state exploration algorithm inside an inductive theorem prover; forward to tackle higher level of abstraction. The work can be extended to implement a complete high level model checking in HOL based on our infrastructure. Including the definition of each $\mathcal{L}_{MDG}$ [14] related algorithm; as a tactic. Another, future work is to conduct a complete system level case study to measure the capabilities of this approach and to ensure that our approach does not create an unacceptable penalty in terms of the performance of the model checker; due to the additional theorem proving overhead.

References

1. S. Abed, O. Ait Mohamed, and G. Al Sammane. On the embedding and verification of multiway decision graph in HOL theorem prover. Technical Report 2007–1–Abed, ECE Department, Concordia University, Montreal, Canada. http://users.encs.concordia.ca/∼s_abed/Technical_Report_2007-1-Abed.pdf, February 2007.
2. O. Ait-Mohamed, X. Song, and E. Cerny. On the non-termination of MDG-based abstract state enumeration. *Theoretical Computer Science*, 300:161–179, 2003.
3. Hasan Amjad. Programming a symbolic model checker in a fully expansive theorem prover. In *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics*, volume 2758 of *Lecture Notes in Computer Science*, pages 171–187. Springer-Verlag, 2003.
4. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.
5. F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny. Multiway decision graphs for automated hardware verification. In *Formal Methods in System Design*, volume 10, pages 7–46, February 1997.
6. M. Gordon. Reachability programming in HOL98 using BDDs. In *TPHOLs*, pages 179–196, 2000.
7. M. Gordon. Programming combinations of deduction and BDD-based symbolic calculation. *LMS Journal of Computation and Mathematics*, 5:56–76, August 2002.
8. John Harrison. Binary decision diagrams as a HOL derived rule. *The Computer Journal*, 38:162–170, 1995.
9. S. Kort, S. Tahar, , and P. Curzon. Hierarchal verification using an MDG-HOL hybrid tool. *International Journal on Software Tools for Technology Transfer*, 4(3):313–322, May 2003.
10. T. Mhamdi and S. Tahar. Providing automated verification in HOL using MDGs. In *Automated Technology for Verification and Analysis*, pages 278–293, 2004.
11. Verification of MDG Algorithms. http://users.encs.concordia.ca/∼s_abed/project.htm.
12. V. Ortner and N. Schirmer. Verification of BDD normalization. In *TPHOLs*, pages 261–277, 2005.
13. K. N. Verma, J. Goubault-Larrecq, S. Prasad, and S. Arun-Kumar. Reflecting BDDs in Coq. In *Proc. 6th Asian Computing Science Conference (ASIAN'2000), Penang, Malaysia, Nov. 2000*, volume 1961, pages 162–181. Springer, 2000.
14. Y. Xu, X. Song, E. Cerny, and O. Ait Mohamed. Model checking for a first-order temporal logic using multiway decision graphs (mdgs). *The Computer Journal*, 47(1):71–84, 2004.