Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 246 (2009) 3–26
www.elsevier.com/locate/entcs

LCF-style Platform based on Multiway Decision Graphs

Sa'ed Abed and Otmane Ait Mohamed [1]

Department of Electrical and Computer Engineering
Concordia University
Montreal, Canada

Abstract

The combination of state exploration approach (mainly model checking) and deductive reasoning approach (theorem proving) promises to overcome the limitation and to enhance the capabilities of each. In this paper, we are interested in defining a platform for Multiway Decision Graphs (MDGs) in LCF-style theorem prover. We define a platform to represent the MDG operations: conjunction, disjunction, relational product and prune-by-subsumption as a set of inference rules. Based on this platform, the reachability analysis is implemented as a conversion that uses the MDG theory within the HOL theorem prover. Finally, we present some experimental results to show the performance of the MDG operations of our platform.

Keywords: Multiway Decision Graphs, LCF-Style, Inference Rules, Performance

1 Introduction

Hardware and software systems are very often critical in the sense that their failure can have considerable human/economical consequences. Therefore, there is a real need of methods for hardware and software development to ensure that those systems are secure and reliable. This includes verification techniques to detect automatically defective behavior of the system and to check that the implementation of a system conforms to its specification (to establish its correctness).

Formal verification of hardware and software systems has gained more attention in both academic world and computer industry since the famous Pentium bug in 1994. The two main approaches of formal verification are based, respectively, on state exploration [17] (mainly model checking and equivalence checking) and deductive reasoning (theorem proving). These two approaches have complementary strengths and weaknesses. The gain obtained in hardware verification has led to

[1] Email: {s_abed,ait}@ece.concordia.ca

doi:10.1016/j.entcs.2009.07.012

the evolution of software verification to achieve similar profits in the context of algorithm and code correctness.

Model checking models the system as a finite-state system and automatically decides if a temporal property holds in the desired states of the system. Furthermore, model checking can produce a counterexample when the property does not hold, which can be very important for the understanding 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.

Theorem proving models the system as a theorem in a mathematical logic and the desired properties are proved by formal derivations. Based on first order and higher order logic, these theorem provers are known for their abilities to express relationships over unbounded data structures. Therefore, theorem proving tools are not sensitive to states explosion problem when used to reason formally about such data and relationships. Yet, they can handle complex systems but requires skilled manual guidance for verification and human insight for debugging. Unfortunately, if the property fails to hold, some theorem proving tools are able to give counterexamples.

An increasing attention has been focused on combining these two approaches to overcome the limitations and to enhance the capabilities of each.

Moreover, the growing importance of model checking in hardware verification and the difficulty of producing correct software are driving a growing interest in the application of model checking to software. This leads to many challenges of scientific and practical interest in the core of model checking technology. However, the transfer of this technique was very slow due to the state explosion problem and the semantics gap between the software developers and the verification tools. Two approaches can be used to tackle these problems: developing new dedicated techniques for software. The advantage of this approach is that the difficulty of applying software model checking is addressed directly [21]. The other approach is the integration of existing tools by using high (abstract) modeling techniques. The advantage of this approach is that the reusing an existing tool and therefore the effort is oriented towards the integration part. The integration of Multiway Decision Graph (MDG) with Bandera framework [8] is a good example of this approach [18].The authors introduced a schema for translating the intermediate representation of a JAVA program using the Bandera framework to the language of the MDG model checking.

In this paper, we focus on the Multiway Decision Graphs (MDGs) [9]. MDG generalizes ROBDD to represent and manipulate a subset of first order logic formula 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 rise the abstraction level of what can be verified using a state exploration within a theorem prover. Furthermore, a verification based on abstract-implicit-state-enumeration can be carried out independently of datapath

width, substantially lessening the state explosion problem.

Our approach is based on embedding the Directed Formulae (DFs): an alternative vision for MDGs in terms of logic and set theory [3], in LCF-style interactive theorem prover as inference rules. We define a platform to represent the MDG operations: conjunction, disjunction, relational product and prune-by-subsumption as a set of inference rules. Based on this platform, the reachability analysis is implemented as a conversion that uses the MDG theory within the HOL theorem prover. Thus, by representing the primitive MDG operations as inference rules added to the core of the theorem prover, we can model the execution of a model checker for a given property as well as raise the security of our platform. Finally, we present some experimental results to show the performance of our platform. The obtained results show that this platform offers a considerable gain in terms of automation without sacrificing CPU time and memory usage. The performance penalty is compensated by the secure and correct infrastructure offered by the HOL theorem Prover.

The paper is organized as follows: Section 2 reviews some related work to our area. Section 3 gives some preliminaries on MDG and HOL systems, respectively. Section 4 explains the inference rules of well-formedness conditions. Section 5 presents the inference rules representing the basic MDG operations. Section 6 introduces the reachability analysis tactic in HOL and summarizes some experimental results and statistics to show the performance of our platform. Finally, Section 7 concludes the paper and gives some future research directions.

2 Related Work

The development of software/hardware verification methods allowing to handel many technological challenges. Both static analysis and model checking research communities are concerned with these challenges to verify properties of software and hardware systems. The main concepts used are based on transition systems, fixpoint computations, reachability analysis, etc. On the other hand, large number of verification techniques are in use as stand alone or in combination with one another [10]. The best advances come from a combination of techniques from different research areas and hence our research is motivated toward this direction.

Gordon integrated the BDD based verification system BuDDy (BDD package implemented in C) into HOL by implementing BDD-based verification algorithms inside HOL, the embedding is built on top of provided primitives. 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 [12,13].

In [15], 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 [4] demonstrated how BDD based symbolic model checking algorithms for the propositional $\mu - calculus$ (L_μ) can be embedded in HOL theorem prover.

This approach allows results returned from the model checker to be treated as theorems in HOL. By representing primitive BDD operations as inference rules added to the core of the theorem prover, the execution of a model checker for a given property is modeled as a formal derivation tree rooted at the required property. These inference rules are hooked to a high performance BDD engine [13] which is external to the theorem prover. Thus, the HOL logic is extended with these extra primitives. Empirical evidence suggests that the efficiency loss in this approach is within reasonable bounds. The approach still leaves results reliant on the soundness of the underlying BDD tools. A high assurance of soundness is obtained at the expenses of some efficiency. Therefore, the security of the theorem prover is compromised only to the extent that the BDD engine or the BDD inference rules may be unsound.

In fact, while BDDs are widely used in state exploration methods, they can only represent Boolean formulae. Our work deals with MDGs rather than BDDs, since MDGs subsume BDDs while accommodating abstract data and uninterpreted function symbols. So we can expect more level of abstraction and more compact representation. Mhamdi and Tahar [19] follow a similar approach to the BuDDy work [13]. The work builds on the MDG-HOL [16] 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. In contrast, their work linked HOL and an external MDG library, while we provide a set of inference rules to represent the MDG operations. Later, these operators can be used as an infrastructure for MDG model checking. Perhaps the experience gained from the work described here will inform the design of a second generation tool integration platform supporting a spectrum from loose to tight integration of external tools.

3 Preliminaries

In this Section, we give a brief introduction to the MDG system as well as to the HOL theorem prover. The intent is to familiarize the reader with the main ideas and notations that are used in the rest of the paper.

3.1 Multiway Decision Graphs

MDGs subsume the class of Bryant's (ROBDD) [5] 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)[9]. The internal nodes are labeled by terms, and the edges issuing from an internal node 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 are used to represent control signals.

- Abstract sort: has no enumeration available. It uses first order terms to represent data signals.

MDGs are canonical representations, 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).

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 [22]. The underline is used to differentiate between the concrete and abstract variables.

Sort $\mathcal{S}$	::=	$S \mid \underline{S}$
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 (individuals), abstract variables, concrete variables and function symbols. DFs are always disjunctions of conjunctions of equations or $\top$ (true) 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. Equation can be an equality of concrete terms and an individual constant, equality of a concrete variable and an individual constant, or equality of an abstract variable and an abstract term.

MDGs provide efficient representation to a class of well-formed first-order formulas defined on well-typed equations. A well-typed equation is an expression $A_1 = A_2$, where A_1 and A_2 are terms of the same sort. 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 [9] such that:

(i) 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;

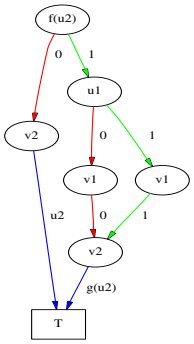
(ii) In each disjunct, the LHSs of the equations are pairwise distinct; and

(iii) 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 $v \in V$).

Intuitively, in a DF of type $U \rightarrow V$, the U variables play the role of independent variables(secondary variables), the V variables play the role of dependent variables (primary variables), and the disjuncts enumerate possible cases. In each disjunct, the equations of the form $u = a$ and $A = a$ specify a case in terms of the U variables, while the other equations specify the values of (some of the) V variables in that case. The cases need not be mutually exclusive, nor exhaustive.

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 g is an abstract function symbol of type $wordn \rightarrow wordn$ and f is a cross-operator of type $wordn \rightarrow bool$. Then, the Figure below shows the MDG representing this example as well as its corresponding DF formula.



$$((f(u_2) = 0) \wedge (v_2 = u_2)) \vee$$
$$((f(u_2) = 1) \wedge (u_1 = 0) \wedge (v_1 = 0) \wedge (v_2 = g(u_2))) \vee$$
$$((f(u_2) = 1) \wedge (u_1 = 1) \wedge (v_1 = 1) \wedge (v_2 = g(u_2)))$$

The MDG model checking is based on an abstract implicit state enumeration.

The system is expressed as an Abstract State Machine (ASM) and the properties to be verified are expressed by formulae in $\mathcal{L}_{MDG}$ [22] language. The MDG operations and verification procedures are packaged as a tool and implemented in Prolog-style hardware description language called (*MDG-HDL*) [7,23], which supports structural specification, behavioral specification or a mixture of both.

3.2 The HOL Theorem Prover

The HOL system is an LCF [11] (Logic of Computable Functions) style proof system. Originally intended for hardware verification, HOL uses higher-order logic to model and verify variety of applications in different areas; serving as a general purpose proof system. We cite for example: reasoning about security, verification of fault-tolerant computers, compiler verification, program refinement calculus, software verification, modeling, and automation theory.

HOL provides a wide range of proof commands, rewriting tools and decision procedures. The system is user-programmable which allows proof tools to be developed for specific applications; without compromising reliability [14]. The basic interface to the system is a Standard Meta Language (SML) interpreter. SML is both the implementation language of the system and the Meta Language in which proofs are written. Proofs are input to the system as calls to SML functions. The HOL system supports two main different proof methods: forward and backward proofs in a natural-deduction style calculus.

Theorems in HOL are represented by values of the ML abstract type **thm**. There is no way to construct a theorem except by carrying out a proof based on the primitive inference rules and axioms. HOL has many built-in inference rules and ultimately all theorems are proved in terms of the axioms and basic inferences of the calculus. By applying a set of primitive inference rules, a theorem can be created. Once a theorem is proved, it can be used in further proofs without recomputation of its own proof.

HOL also has a rudimentary library facility which enable theories to be shared. This provides a file structure and documentation format for self contained HOL developments. Many basic reasoners are given as libraries such as mesonLib, bossLib, and simpLib. These libraries integrate rewriting, conversion and decision procedures to free the user from performing low-level proof.

4 Well-formedness Inference Rules

Using HOL recursive datatypes, the MDG sort is either concrete or abstract sort. 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
```

Similarly, constants 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. We use the `Ind_Cons` and `Gen_Cons` constructors to declare constants in HOL. Also a variable (abstract or concrete) is identified by its name and sort. In our embedding, an abstract variable is declared using `Abst_Var` constructor and the `Conc_Var` constructor is used to declare a concrete variable.

Functions can be either abstract or cross-functions. Cross-functions are those that have at least one abstract argument. Note that concrete functions are not used since they can be eliminated by case splitting. If *eqz_Fun* is a cross-function takes an abstract variable m as an input and produces a concrete output of sort *bool*, then, *eqz_Fun* is defined as:

```
⊢def  m = Abst_Var "m" wordn
⊢def  eqz_Fun = Cross_Function "eqz_Fun" [^m] bool
```

The type definition of a directed formula can be given as follows:

```
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_Var => '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
```

where `DF1, DISJ, CONJ1, Eqn, CONJ` are distinct constructors and the constructors `EQUAL1, EQUAL2, EQUAL3, EQUAL4, EQUAL5` are used to define the atomic equation. The type definition package returns a theorem which characterizes the type `D_F` and allows reasoning about this type. `Abs_var, Conc_var` are defined as datatype to represent Abstract variables and Concrete variables. 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.

In order to keep a DF formulae in its logical format, i.e. disjunctions of conjunctions, we devised a set of inference rules which decides if a given DF is well formed or not. The recursive data type package automatically returns the following theorems which characterize each condition separately. In Table 1, we formalized these conditions as a set of rules since it is more adequate to DF, and independent from the logic of theorem prover. We translate these inference rules as theorems in HOL.

The well-formedness conditions can be summarized as:

- The first condition is satisfied by construction following the DF syntax. The axiom `WF_E1` represents the equality between a concrete variable and a concrete in-

Table 1
Well-Formedness (WF) Inference Rules

$$\texttt{WF\_True:}\ \frac{-}{WF(T)}$$

$$\texttt{WF\_False:}\ \frac{-}{WF(F)}$$

$$\texttt{WF\_E1:}\ \frac{-}{WF(\underline{V} = \underline{C})}$$

$$\texttt{WF\_E2\_E4\_E5:}\ \frac{-}{WF(V = A)}\ ;\ (A \in \mathcal{T}(\mathcal{F}, \mathcal{X}))$$

$$\texttt{WF\_E3:}\ \frac{-}{WF(\underline{A} = \underline{C})}\ ;\ (A \in \mathcal{T}(\mathcal{F}, V))$$

$$\texttt{WF\_Conj:}\ \frac{WF(Eq_1)\quad WF(Eq_2)\quad (LHS(Eq_1) \neq LHS(Eq_2))}{WF(Eq_1 \wedge Eq_2)}$$

$$\texttt{WF\_Disj:}\ \frac{WF(Conj_1)\ \ WF(Conj_2)\ \ (Abst\_Var(Conj_1) = Abst\_Var(Conj_2))}{WF(Conj_1 \vee Conj_2)}$$

dividual constant. Axiom `WF_E2_E4_E5` shows the equality of an abstract variable and an abstract term (abstract variable, abstract generic constant and abstract function symbol). Finally, axiom `WF_E3` expresses the equality of concrete term and concrete individual constant.

- The second condition requires two well-formed equations and the Left Hand Side (LHS) of each equation should not be equal. The rule `WF_Conj` states the correctness related to this condition.
- The assumptions needed for the third condition are: two well-formed conjuncts and the abstract variables of each conjunct should be equal. The `WF_Disj` rule represents the correctness related to this last condition.

We have implemented a HOL tactic `Is_Well_Formed_DF` (conversion tactic) to automatize the checking of well-formedness conditions. This tactic returns whether a given DF is well-formed or not [1].

5 MDGs Operation Inference Rules

The MDG-HOL platform provides all the necessary infrastructure (data structure + algorithms) to define an abstract state exploration in the HOL theorem prover. For this mean, the DF vocabulary and well-formedness conditions are defined in [1] based on the directed formulae syntax. Also, we provide formal definitions of the MDG basic operations as inference rules within HOL. In this Section, we describe the MDG operations in terms of inference rules. Then, we present the reachability conversion and provide some experimental results to show the performance of the MDG operations (mainly PbyS operation) and the MDG-HOL platform.

5.1 The 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 produces a DF R = **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:

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

The method for computing the conjunction of two DFs is applicable when the sets of primary variables of the two DFs are disjoint. The resulting DF has primary variables that are among the primary variables of the conjuncts, including all abstract variables that have 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.

As shown below, we formalized the conjunction operation of two DFs as inference rules. Axioms (R1 to R4) represent the terminal case i.e. when one of the two DFs is TRUE or FALSE. These rules form a proof system in analogy to logical rules. The horizontal line reads implies. Thus rules represent logical truths. It follows that rules with nothing above the line are axioms since they always hold.

`Terminal DF Axioms:`

(R1) $\dfrac{}{CONJ\_ALG(TRUE, df) = df}$ (R2) $\dfrac{}{CONJ\_ALG(df, TRUE) = df}$

(R3) $\dfrac{}{CONJ\_ALG(FALSE, df) = FALSE}$ (R4) $\dfrac{}{CONJ\_ALG(df, FALSE) = FALSE}$

The result of the operation must be a well-formed DF representing the conjunction of df1 and df2. Thus, it suffices to eliminate RHS of Eq1 by substitution for the LHS in Eq2 or replacing the secondary occurrences of RHS in Eq2 with the respective terms (LHS of Eq1). The function `SUBST(Eq1,Eq2)` represents the substitution using λ abstraction when it is applicable. Axiom R5 describes the conjunction between two equations. The `CONJ_ALG` function computes the conjunction of two well formed DFs [2].

`DF Equation Axioms:`

(R5) $\dfrac{}{CONJ\_ALG(Eq1, Eq2) = SUBST(Eq1, Eq2)}$

where $SUBST(Eq1, Eq2) = \begin{cases} (\lambda(L(Eq2)).Eq1)R(Eq2) & \text{if } ((R(Eq1) \neq L(Eq2)) \\ (\lambda(L(Eq1)).Eq2)R(Eq1) & \text{if } ((L(Eq1) \neq R(Eq2)) \\ Eq1 \wedge Eq2 & \text{otherwise} \end{cases}$

and $L(x = y) = x, \quad R(x = y) = y, \qquad Abs(Eq1) = L(Eq1) \in Abstract\ Variable$

Rules R6 and R7 represent the distribution relation rules and show the conjunction between an equation *Eq* and a set of equations *conj*, respectively. Note that *conj* is defined as the conjunction of an equation *Eq* and a set of equations *conj'*. R6 is carried out by computing the conjunction (*R1'*: conjunction of two equations) and (*R2'*: conjunction of *Eq* and *conj'*). Rule R8 computes the conjunction between *conj1* and *conj2*. Similarly, rules R9 and R10 compute the conjunction between a *conj* and a *df* while rule R11 determines the conjunction over two DFs. Each of these rules calls recursively the previous rules until reach an axiom.

`Conjunct and DF Rules:`

$$\texttt{(R6)}\ \frac{CONJ\_ALG(Eq1, Eq) \Rightarrow R1' \quad CONJ\_ALG(Eq1, conj2') \Rightarrow R2'}{CONJ\_ALG(Eq1, conj2) \Rightarrow R1' \wedge R2'}$$

where

$conj1 = Eq \wedge conj1'$, $conj2 = Eq \wedge conj2'$, $df1 = conj \vee df1'$, $df2 = conj \vee df2'$, and

$SUBST(Eq1, conj2) = SUBST(Eq1, Eq) \wedge SUBST(Eq1, conj2')$

$$\texttt{(R7)}\ \frac{CONJ\_ALG(Eq, Eq2) \Rightarrow R1' \quad CONJ\_ALG(conj1', Eq2) \Rightarrow R2'}{CONJ\_ALG(conj1, Eq2) \Rightarrow R1' \wedge R2'}$$

$$\texttt{(R8)}\ \frac{CONJ\_ALG(Eq1, Eq2) \Rightarrow R1' \quad CONJ\_ALG(conj1', conj2') \Rightarrow R2'}{CONJ\_ALG(conj1, conj2) \Rightarrow R1' \wedge R2'}$$

$$\texttt{(R9)}\ \frac{CONJ\_ALG(conj1, conj2) \Rightarrow R1' \quad CONJ\_ALG(conj1, df2') \Rightarrow R2'}{CONJ\_ALG(conj1, df2) \Rightarrow R1' \vee R2'}$$

$$\texttt{(R10)}\ \frac{CONJ\_ALG(conj1, conj2) \Rightarrow R1' \quad CONJ\_ALG(df1', conj2) \Rightarrow R2'}{CONJ\_ALG(df1, conj2) \Rightarrow R1' \vee R2'}$$

$$\texttt{(R11)}\ \frac{CONJ\_ALG(conj1, df2) \Rightarrow R1' \quad CONJ\_ALG(df1', df2) \Rightarrow R2'}{CONJ\_ALG(df1, df2) \Rightarrow R1' \vee R2'}$$

As those rules are proved correct, the strategy of applying them starts from rule (R11) to find the conjunction of two DFs by recursion over DF (taking the first disjunct with the second DF and then recursively calling the other rules). The recursion step terminates when executing one of the axioms based on the equation type. Note that the depth of the directed formula is reduced each time a rule is called, and hence the conjunction algorithm terminates.

The correctness proof of the conjunction operation is given in [2]. The result of the `CONJ_ALG` operation must be well-formed DF representing the conjunction of two dfs (df1 and df2) as shown in Theorem 1:

Theorem 5.1 *Conjunction well-formedness*

```
⊢  ∀df1 df2. ∃L. Is_Well_Formed_DF df1 ∧ Is_Well_Formed_DF df2 ∧
(ORDER_LIST df1 df2 = L) ⟹  Is_Well_Formed_CONJ df1 df2 L
```

Proof (Sketch) The goal is to prove that the result of the embedded conjunction operation in HOL is well-formed. The proof is conducted by structural induction on df1 and df2 and rewriting rules. □

Similar theorems were proved for the RelP, disjunction and PbyS operations.

The relational product operation (RelP) is used to compute the sets of states reachable in one transition from one set of states. It combines conjunction and existential quantification. The inference rules formalization of the RelP operation are based on the rules of the conjunction operation and hence will not be repeated again. This simplifies the formalization and shows the reusability of our work.

5.2 The Disjunction Operation

The disjunction operation takes as inputs two DFs P_i, $1 \leq i \leq 2$, of types $U_i \rightarrow V$, and produces a DF $R = \textbf{Disj}\ (\{P_i\}_{1 \leq i \leq 2})$ of type $(\bigcup_{1 \leq i \leq 2} U_i) \rightarrow V$ such that:

$$(2) \quad \models R \Leftrightarrow (\bigvee_{1 \leq i \leq 2} P_i)$$

The operation requires that all the P_i, $1 \leq i \leq 2$, have the same set of abstract primary variables. If two DFs P_1, P_2 do not have the same set of abstract primary variables, then there is no DF R such that $\models R \Leftrightarrow (P_1 \vee P_2)$.

The axioms and rules shown in Table 2 express the logical semantics of the disjunction operation in terms of HOL.

The `DISJ_ALG` function computes the disjunction of two well formed DF [2] and the `D_INFR` function computes the disjunction of an equation and a conjunct considering all cases.

5.3 Pruning by Subsumption (PbyS) Operation

The PbyS operation represents the core of the reachability analysis algorithm. It 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 $R = \textbf{PbyS}\ (P, Q)$ of type $U \rightarrow V_1$ derivable from P by *pruning* (i.e. by removing some of disjoints) such that:

$$(3) \quad \models R \vee (\exists E)Q \Leftrightarrow P \vee (\exists E)Q$$

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 (3), 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 $(\exists E)Q$.

The inference rules describing the PbyS operation are shown in Table 3. Axioms R1 to R4 represent terminal DF cases. Axioms R5 to R7 show the PbyS operation for the case of two equations, equation and conjunct and two conjuncts, respectively. The function `PbyS_ALG` function computes the pruning by subsumption of two well formed DF [2] and the `Prune_Eq` function checks if an equation or a conjunct exists

Table 2
Inference Rules for Disjunction Operation

Terminal DF Axioms:

$$\text{(R1)}\ \frac{-}{DISJ\_ALG(TRUE, df) = TRUE} \qquad \text{(R2)}\ \frac{-}{DISJ\_ALG(df, TRUE) = TRUE}$$

$$\text{(R3)}\ \frac{-}{DISJ\_ALG(FALSE, df) = df} \qquad \text{(R4)}\ \frac{-}{DISJ\_ALG(df, FALSE) = df}$$

DF Equation Axioms:

$$\text{(R5)}\ \frac{-}{DISJ\_ALG(Eq1, Eq2) = D\_INFR(Eq1, Eq2)}$$

$$\text{(R6)}\ \frac{-}{DISJ\_ALG(Eq1, conj2) = D\_INFR(Eq1, conj2)}$$

where

$$conj1 = Eq1 \wedge conj1',\ conj2 = Eq2 \wedge conj2',\ df1 = conj1 \vee df1',\ df2 = conj2 \vee df2',$$

$$D\_INFR(Eq1, conj2) = \begin{cases} FALSE & \text{if } (Abs(Eq1) \neq Abs(conj2)) \\ conj2 & \text{if } (Eq1 = Eq2) \\ Eq1 \wedge conj2 & \text{if } (L(Eq1) < L(Eq2)) \\ Eq2 \wedge D\_INFR(Eq1, conj2') & \text{if } (L(Eq1) > L(Eq2)) \\ (Eq1 \vee Eq2) \wedge conj2' & \text{otherwise} \end{cases}$$

$L(x = y) = x$, $<:$ *order*, and $Abs(Eq1) = L(Eq1) \in Abstract\ Variable$

Conjunct and DF Rules:

$$\text{(R7)}\ \frac{DISJ\_ALG(Eq1, Eq2) \Rightarrow R1' \quad DISJ\_ALG(conj1', conj2') \Rightarrow R2'}{DISJ\_ALG(conj1, conj2) \Rightarrow R1' \wedge R2'}$$

$$\text{(R8)}\ \frac{DISJ\_ALG(conj1, conj2) \Rightarrow R1' \quad DISJ\_ALG(conj1, df2') \Rightarrow R2'}{DISJ\_ALG(conj1, df2) \Rightarrow R1' \vee R2'}$$

$$\text{(R9)}\ \frac{DISJ\_ALG(conj1, conj2) \Rightarrow R1' \quad DISJ\_ALG(df1', conj2) \Rightarrow R2'}{DISJ\_ALG(df1, conj2) \Rightarrow R1' \vee R2'}$$

$$\text{(R10)}\ \frac{DISJ\_ALG(conj1, df2) \Rightarrow R1' \quad DISJ\_ALG(df1', df2) \Rightarrow R2'}{DISJ\_ALG(df1, df2) \Rightarrow R1' \vee R2'}$$

in the other equation or conjunct. Similarly, the `Prune_conj` function checks if the conjunct exists in the other conjunct considering all cases. Rule R8 computes the PbyS of a conjunct and a Df and rule R9 computes the PbyS of two DFs.

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 such as RelP and disjunction operations. The embedding of MDG syntax and the verification of MDG operations sums up to several thousand lines of HOL codes. The complexity of the proof is related mainly to the MDG structure, and the recursive definitions of MDG operations.

Table 3
Inference Rules for PbyS Operation

Terminal DF Axioms:

(R1) $$\frac{-}{PbyS\_ALG(TRUE, df) = FALSE}$$ (R2) $$\frac{-}{PbyS\_ALG(df, TRUE) = FALSE}$$

(R3) $$\frac{-}{PbyS\_ALG(FALSE, df) = FALSE}$$ (R4) $$\frac{-}{PbyS\_ALG(df, FALSE) = df}$$

Equation Axioms:

(R5) $$\frac{-}{PbyS\_ALG(Eq1, Eq2) = Prune\_Eq(Eq1, Eq2)}$$

(R6) $$\frac{-}{PbyS\_ALG(Eq1, conj2) = Prune\_Eq(Eq1, conj2)}$$

where

$$conj1 = Eq1 \wedge conj1',\ conj2 = Eq2 \wedge conj2',\ df1 = conj1 \vee df1',\ df2 = conj2 \vee df2',$$

$$Prune\_Eq(Eq1, conj2) = \begin{cases} FALSE & \text{if } (Eq1 \in conj2) \vee (Abs(Eq1) \in Abs(conj2)) \\ Eq1 & \text{otherwise} \end{cases}$$

$L(x = y) = x$, $<:$ *order*, and $Abs(Eq1) = L(Eq1) \in Abstract\ Variable$

(R7) $$\frac{-}{PbyS\_ALG(conj1, conj2) = Prune\_conj(conj1, conj2)}$$

where

$$Prune\_conj(conj1, conj2) = \begin{cases} FALSE & \text{if } Prune\_Eq(Eq1, conj2) \vee \\ & \quad Prune\_conj(conj1', conj2) = FALSE \\ conj1 & \text{otherwise} \end{cases}$$

Conjunct and DF Rules:

(R8) $$\frac{PbyS\_ALG(conj1, conj2) \Rightarrow R1' \quad PbyS\_ALG(conj1, df2') \Rightarrow R2'}{PbyS\_ALG(conj1, df2) \Rightarrow R1' \wedge R2'}$$

(R9) $$\frac{PbyS\_ALG(conj1, df2) \Rightarrow R1' \quad PbyS\_ALG(df1', df2) \Rightarrow R2'}{PbyS\_ALG(df1, df2) \Rightarrow R1' \vee R2'}$$

5.3.1 The PbyS Performance

In this section, we present the performance of the PbyS operation. The results are carried out using a Sun server with Solaris 5.7 OS and 6 GB memory. We analyze the required time for generating a result from PbyS by applying it over two well-formed DFs. One DF has a size of 182 disjuncts with a 32 equations in each disjunct. For the results given in Figure 1, in each run we increase the size of the disjunct and measure the execution time.

As a result, the execution time is increased when the number of disjuncts is increased. This is due to the increase in the DF size in terms of the number of

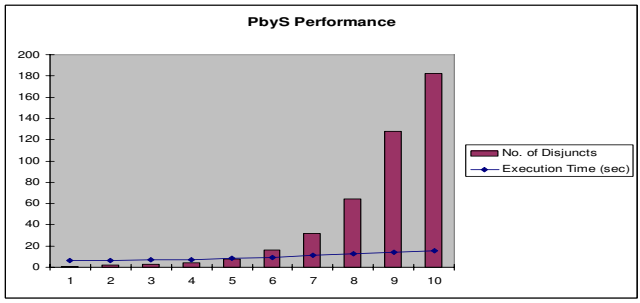


Fig. 1. The PbyS Performance

disjuncts.

Figure 1 shows the results of the execution time vs. number of disjuncts. We note that the execution time is almost linear which emphasizes the effectiveness and the powerful of our embedding. The average execution time is 10.0. We consider this time is normal because of the overhead of the theorem prover. However, a huge investment in time should be spent in developing the theory and proving the necessary theorems in theorem provers.

6 The Reachability Analysis

Here, we present a HOL formalization of the MDG reachability analysis. This formalization is based on our embedding of MDG syntax and operations in HOL. First, we will review the MDG reachability analysis [9]; followed by its definition in HOL along with a discussion on the technical challenges. Finally, we will use the MIN-MAX design as an illustrative example for our reachability analysis embedding.

6.1 Reachability Analysis Algorithm

The presence of uninterpreted symbols in the logic means that we must distinguish between a state machine M and its abstract description D in the logic. We refer to state machines whose transition relation, output relation, and the set of initial states are given by DFs, or equivalently MDGs, as *Abstract State Machine* (ASMs) as defined in [9].

Definition 6.1 An abstract description of a state machine M is a tuple $D = (X, Y, Z, Y\prime, IS, Tr, Or)$, where:

- X : finite set of input variables,
- Y : finite set of state variables,
- Z : finite set of output variables,
- $Y\prime$: finite set of next-state variables,
- IS : MDG of type $U_0 \rightarrow Y$, where U_0 is a set of disjoint abstract variables, IS is the abstract description of the set of initial states,

- $Tr : MDG$ of type $X \cup Y \rightarrow Y\prime$. Tr is the abstract description of the transition relation,
- $Or : MDG$ of type $X \cup Y \rightarrow Z$. Or is the abstract output relation.

Algorithm 1 shows how the analysis of the reachable states of M can be performed based on the abstract description D.

Algorithm 1 MDG Reachability Analysis

1: $R := IS$;
2: $Q := IS$;
3: $i := 0$;
4: **while** $Q \neq F$ **do**
5: $\quad i := i + 1$;
6: $\quad IN := new\_inputs(i)$; – `Produce new inputs`
7: $\quad NS := next\_states(IN, Q, Tr)$; – `Compute next state`
8: $\quad Q := frontier(NS, R)$; – `Set difference`
9: $\quad R := union(R, Q)$; – `-- Merge with set of states reached previously`
10: **end while**

Lines 1-3 initialize the algorithm by constructing the initial MDG structure. In line 4-10, the set of reachable states is computed within the while loop. The while loop terminates when the frontier set (Q) becomes empty (F). In line 6, a new MDG input is produced. In line 7, the function *next_state* computes the next state using the RelP operation which takes as assignment the MDGs representing the set of inputs, the current state and the transition relation, respectively. The function $frontier$, in line 8, computes the set difference using the PbyS operation. This operation approximates the set difference between the newly reachable state in the current iteration from the reachable state in the first iteration. Finally, in line 9, the set of all reachable states so far is computed.

6.2 Formalization of Reachability Analysis

We show here the steps to formalize the set of reachable states of an abstract state machine in HOL. The important difference is that we are using our embedded DF operators at a higher 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.

Since reachability analysis may not terminate in general, it's impossible to prove a general theorem which states the existence of a fixpoint for all designs. However, we defined a conversion which returns a goal to be proven interactively using induction for a given design (DF). If we succeed to prove the goal, then we can conclude that the reachability analysis terminates. The general fixpoint goal has the following format:

```
∃n0. ∀n. (n>n0) ⟹ (Re_An (SUC n) I Q Tr E Ren L R = Re_An n I Q Tr E Ren L R)
```

where `n0` is the number of iterations needed to reach a fixpoint and the `Re_An`

function represents the MDG reachability analysis with the following parameters: the set of input variables I, the set of initial states Q, the transition relation Tr, the set of variables to be quantified E, the state variables to be renamed Ren, the order list L and the initial reachable states R.

The function `Re_An` is defined in HOL by calling the recursive function `RA_n` with the design parameters. The function `RA_n` represents the set of reachable states and includes the following functions:

- The `Next_State` function: computes the set of next states reached from a set of given states with respect to the transition relation of the design. The result is obtained using the DF relational product operator `RelP(Q,Tr)`:

```
⊢def (Next_State I I_F Q Q_F Tr Tr_F Tr_A E Ren L =
      Rename (EXIST_QUANT (rep_list (TAKE_HD (DF_CONJ
       I (rep_list (TAKE_HD (DF_CONJ Q Tr (union Q_F Tr_F) L)))
       (union (union I_F Q_F) Tr_F) L))) E) Ren)
```

- The `Frontier_Step` function: checks if all the states reachable by the machine are already visited. The is done by using the `PbyS(RelP(Q,Tr),R)` operator:

```
⊢def (Frontier_Step I I_F Q Q_F Tr Tr_F Tr_A E Ren L R R_F R_A =
      DF_PbyS (Next_State X X_F Q Q_F Tr Tr_F Tr_A E Ren L) R
               (union Tr_F R_F) Tr_A R_A L)
```

  If the result is the empty set, then the reachability analysis terminates. Otherwise, it returns the new frontier set.

- The `Union_Step` function: merges the output of `Frontier_Step` with the set of states reached previously using the `PbyS` and disjunction operators:

```
⊢def (Union_Step I I_F Q Q_F Tr Tr_F Tr_A E Ren L R R_F R_A =
     rep_list(DF_PbyS R (Frontier_Step I I_F Q Q_F Tr Tr_F Tr_A E Ren L R R_F R_A)
                        (union Tr_F R_F) Tr_A R_A L))
```

Those functions are encapsulated in one function called `Reach_Step` to represent the first iteration of the MDG reachability analysis algorithm:

```
⊢def (Reach_Step I I_F Q Q_F Tr Tr_F Tr_A E Ren L R R_F R_A =
     if (FLAT(Frontier_Step I I_F Q Q_F Tr Tr_F Tr_A E Ren L R R_F R_A)=[]) then
        R
     else
      DF_DISJUNCTION (Union_Step I I_F Q Q_F Tr Tr_F Tr_A E Ren L R R_F R_A)
        (Frontier_Step I I_F Q Q_F Tr Tr_F Tr_A E Ren L R R_F R_A)L)
```

Then, the `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+1) Design_Parameters = RA_n (n) Design_Parameters
```

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 proof of the reachability fixpoint depends on the structure of the design and cannot be considered as a general solution because of the non-termination problem [9]. Like other reachability analysis algorithms that use abstract types and

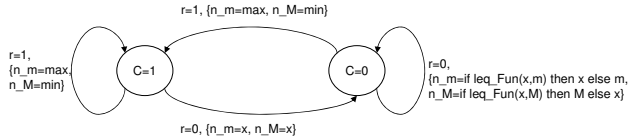


Fig. 2. MIN-MAX State Machine

uninterpreted function symbols, the algorithm may not terminate. Thus, the MDG reachability computation is theoretically unbounded. Meanwhile, several practical solutions have been proposed to solve the non-termination problem. The authors in [3] related the problem to the nature of the analyzed design. Furthermore, they have characterized some mathematical criteria that leads explicitly to non-termination of particular classes of designs.

The reachability analysis conversion is general and can be applied to any DF of a design. What will change is only the DF and the set of initial states, if we consider the order list is given. The conversion shown by Algorithm 2 encapsulates the following steps:

Algorithm 2 Re_An Conversion (I, Q, Tr, E, Ren, L, R)

1: Formalize the design parameters in terms of DF and check for WF: WF(Tr); WF(Q); WF(R)
2: Compute `Reach_Step`.
3: Generate a fixpoint goal of `Re_An`.

The algorithm takes as an input the design parameters. In line 1, we formalize those parameters in terms of DF and then check the well-formedness of all DFs (Tr, Q, R). In line 2, we compute one reachability computational step using the `Reach_Step` function. Finally, in line 3, we generate a fixpoint goal of `Re_An`. The advantage of this approach is that we compute the reachable states for only one iteration and then relying on the induction power in HOL we prove the existence of a fixpoint. However, this fixpoint may not exist for some particular designs. Furthermore, the selection of `n0` is based on the knowledge and the heuristic of the design since the induction is not explicitly identified as illustrated by the case of the MIN-MAX example.

6.3 The MIN-MAX Example

We have chosen a Java implementation of a MIN-MAX design described in [9]. The MIN-MAX state machine shown in Figure 2 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 transition labels specify the conditions under which each transition is taken and an assignment of values to the abstract next state variables n_m and n_M.

The machine stores in m and M, respectively, the smallest and the greatest values presented at the input x since the last reset ($r = 1$). When the machine is reset, m is loaded by the maximal possible value max and M by the minimal possible value min. The min and max symbols are uninterpreted generic constants of sort s. The smallest and greatest values are computed using an operator leq_Fun such that for any two values a and b of sort s, $leq\_Fun(a, b) = 1$ if and only if a is less than or equal to b. The transition relation can be described by a set of individual transition relations, one associated with each next state variable.

The code of the MIN-MAX implementation is used as an input file for Bandera tool. Based on [18] work, we have derived the MDG-HDL model from the BIR specification. The DFs of the individual transition relations of the MIN-MAX design for a particular custom symbol order are:

$$\begin{aligned}
Tr_c &= [((r = 0) \wedge (n_c = 0)) \bigvee \\
&\quad ((r = 1) \wedge (n_c = 1))] \\
Tr_m &= [((r = 0) \wedge (c = 0) \wedge (n_m = m) \wedge (leq_Fun(x, m) = 0)) \bigvee \\
&\quad ((r = 0) \wedge (c = 0) \wedge (n_m = x) \wedge (leq_Fun(x, m) = 1)) \bigvee \\
&\quad ((r = 0) \wedge (c = 1) \wedge (n_m = x)) \bigvee \\
&\quad ((r = 1) \wedge (n_m = max))] \\
Tr_M &= [((r = 0) \wedge (c = 0) \wedge (n_M = x) \wedge (leq_Fun(x, M) = 0)) \bigvee \\
&\quad ((r = 0) \wedge (c = 0) \wedge (n_M = M) \wedge (leq_Fun(x, M) = 1)) \bigvee \\
&\quad ((r = 0) \wedge (c = 1) \wedge (n_M = x)) \bigvee \\
&\quad ((r = 1) \wedge (n_M = min))]
\end{aligned}$$

The DF of the system transition relation Tr is the conjunction of these individual transition relations. The MIN-MAX state machine has two input variables: $I = [[[x; r]]]$, set of initial states:

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

three state variables to be renamed: $Ren = [[c; n\_c]; [m; n\_m]; [M; n\_M]]$, set of variables to be quantified: $E = [r; c; m; M]$, the order list: $L = [r; c; n\_c; m; n\_m; M; n\_M; x; leq\_Fun]$ and the initial reachable state $R = Q$.

Then, we applied the reachability analysis conversion steps mentioned in Algorithm 2:

The first step: formalize the MIN-MAX design in terms of DF and check the well-formedness conditions on (Tr, Q, R). The DF of the system transition relation Tr is the conjunction of these individual transition relations. We illustrate with this

example how the directed formula is defined and how the well-formedness conditions are checked. We just give some of the definitions for concrete and abstract sorts, constants, variables and abstract function, cross-function, equation and disjuncts.

```
⊢def  bool = Conc_Sort "bool" ["0";"1"]
⊢def  wordn = Abst_Sort "wordn"
⊢def  oone = Ind_Cons "1" bool
⊢def  r = Conc_Var "r" bool
⊢def  x = Abst_Var "x" wordn
⊢def  m = Abst_Var "m" wordn
⊢def  n_m = Abst_Var "n_m" wordn
⊢def  leq_Fun = Cross_Fun "leq_Fun" [ "x";"m"] 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) returns the theorem below:

```
⊢  Is_Well_Formed_DF Tr
```

Stating that the directed formula `Tr` is well-formed.

An example of applying (`WF`) inference rules given in Table 1, is presented in Table 4. Since the top symbol is a disjunction then `WF_Disj` rule splits the goal `WF(Tr=`$(eq2 \wedge eq4 \wedge eq11 \wedge eq16) \vee (mdg2 \vee mdg3 \vee mdg4 \vee mdg5 \vee mdg6)$`)` into two subgoals `WF(Tr1=`$(eq2 \wedge eq4 \wedge eq11 \wedge eq16)$`)` and `WF(Tr2=`$mdg2 \vee mdg3 \vee mdg4 \vee mdg5 \vee mdg6)$`)`. $Tr1$ is a conjunct, the `WF_Conj` will be applied until an axiom (final result) is applied.

Table 4
Well-Formed Tr

$$\dfrac{\dfrac{\overset{-}{WF(r = oone)}\ \texttt{WF\_E1}\quad WF(eq4 \wedge eq11 \wedge eq16)\quad Cond2}{WF(eq2 \wedge eq4 \wedge eq11 \wedge eq16)}\ \texttt{WF\_Conj}\quad WF(mdg2 \vee \ldots \vee mdg6)\quad Cond3}{WF((eq2 \wedge eq4 \wedge eq11 \wedge eq16) \vee (mdg2 \vee mdg3 \vee mdg4 \vee mdg5 \vee mdg6))}\ \texttt{WF\_Disj}$$

where:
$eq2 = (r = oone)$
$Cond2 = (LHS(eq2) \neq LHS(eq4) \neq LHS(eq11) \neq LHS(eq16))$
$Cond3 = (Abst\_Var(eq2 \wedge eq4 \wedge eq11 \wedge eq16) = Abst\_Var(mdg2 \vee mdg3 \vee mdg4 \vee mdg5 \vee mdg6))$

The second step: we apply only one `Reach_Step` to compute the next reachable state as explained in Algorithm 2, the reachable states are:

$$\begin{aligned} R1 \quad = \quad & [((c = 0) \wedge (m = x1) \wedge (M = x1)) \bigvee \\ & ((c = 1) \wedge (m = max) \wedge (M = min))] \end{aligned}$$

The third step: the MDG reachability analysis `Re_An` is performed by calling

`RA_n` with the MIN-MAX parameters. `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+1) MinMax_Parameters = RA_n (n) MinMax_Parameters
```

We prove how a fixpoint is reached after `n0` iterations by instantiating the parameters of MIN-MAX. We achieve a fixpoint after three `Reach_Step` calls (`n0`= 2) as shown by the following theorem:

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

where the (t) notation is used in HOL to instantiate the value of the term t. The base step is straightforward and the induction step is carried out by rewriting rules.

Finally, the reachable states at the third iteration are the same as $R2$:

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

6.4 The Platform Performance

We support our platform by experimental results executed on different benchmarks. We consider four cases from the MDG benchmark suites in order to measure the performance of MDG-HOL. The case studies cover two small benchmarks: MIN-MAX and Abstract Counter, one intermediate benchmark: Look-Aside Interface (LA-1) [20], and one large benchmark: Island Tunnel Controller (ITC) [24]. The performance is measured in terms of full reachability analysis for these models. Tables 5 and 6 compare the number of nodes, number of functions, the memory usage, reachability analysis time (RA), and human effort generated by MDG-HOL and FormalCheck (V2.3) [6] model checking, respectively, run on a Sun enterprize server with Solaris 5.7 OS and 6.0 GB memory. The time is measured in FormalCheck by estimating the average time for the set of all properties associated with the design.

Table 5
MDG-HOL Benchmarks

	MDG-HOL				
Example	**No. of Nodes**	**No. of Funcs**	**MEM (MB)**	**RA (sec)**	**Human Effort (H)**
MIN-MAX	54	3	0.533	7	120
Abstract Counter	46	3	0.318	7	120
LA-1	1682	66	0.613	8	216
ITC	118035	27	0.47	9	480

Table 5 shows that the number of nodes and number of functions of the MDG are smaller than its corresponding generated by FormalCheck for small benchmarks (i.e. MIN-MAX and Abstract Counter). This is due to the absence of boolean encoding, i.e. we don't encode the values of model variables. On the other hand, the computation time for the reachability analysis is better in the case of FormalCheck. This is normal because of the overhead of the theorem prover.

Table 6
FormalCheck Benchmarks

	FormalCheck				
Example	**No. of Nodes**	**No. of Funcs**	**MEM (MB)**	**RA (sec)**	**Human Effort (H)**
MIN-MAX	256	6	3.67	6	1
Abstract Counter	128	14	3.43	1	1
LA-1	4096	19	4.02	12	2
ITC	1.76E+12	179	9.07	29	4

As the size of the benchmark increases, the MDG-HOL gives much better results since it does not take a lot of time to load the fixpoint theorem and also the memory usage is negligible as shown in Table 6. However, the number of FormalCheck allocated nodes tends to be greater and hence have a negative impact on computation reachability analysis time and memory usage. The trade off between MDG-HOL and FormalCheck is the time and human effort since it took much more to prove reaching a fixpoint compared to the FormalCheck. This comes from the fact that theorem provers are interactive while model checkers are automatic.

In fact, the performance of the MDG-HOL is considerable, but it cannot replace current model checking tools as it fails to obtain fixpoint proof without major human efforts. However, a huge investment in time should be spent in developing the theory and proving the necessary theorems in theorem provers.

7 Conclusion

MDGs have been proposed to extend BDDs 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 MDGs as formulae in higher order logic using the Directed Formulae notations. The well-formedness conditions and the operations were implemented as a set of inference rules. The ideas described here are intended to establish a correct platform for securely programming new verification algorithms.

Since software and hardware are deployed in many applications, correctness is becoming an important issue. Therefore, software and hardware verification will be a big face for both academic and industry world and hence our research is motivated toward providing the users with a good and a clean logical view based on higher

order logic that supports MDGs. The presented approach is to develop synergies between software and hardware verification concepts, i.e. software verification of MDG tool (data structure + algorithms) which is used for hardware verification in LCF-style theorem prover.

The performance results for the PbyS operation have shown that the execution time of the PbyS operation is almost linear which emphasizes the effectiveness of our embedding. The experimental results based on benchmarks, have shown that the MDG-HOL platform provided a better performance than FormalCheck in terms of time, memory usage, number of nodes, and number of functions especially when the design is growing up. On the other hand, the human efforts are huge compared to FormalCheck. Thus, a complete model checker can be implemented in HOL based on our infrastructure. Including the definition of each $\mathcal{L}_{MDG}$ [22] related algorithm as a tactic in HOL. The model checker will be a complete theory in HOL, but indeed more investigation and formalism is needed to this task. In this context, our reachability conversion can be used to make calls to our defined MDG algorithms, to check wether an L_{MDG} property is valid. Here, we are not reducing the role of the proof expert, but we provide him with an automated conversions that reduces considerably the time he spent. Also, the work can be seen as a formal proof for the MDG model checking approach; verifying a verification system using another verification system.

References

[1] S. Abed and O. Ait Mohamed. Embedding of MDG directed formulae in HOL theorem prover. In *Proc. of MCSEAI'06*, pages 659–664, Agadir, Morocco, December 2006.

[2] S. Abed, O. Ait Mohamed, and G. Al Sammane. Reachability analysis using multiway decision graphs in the HOL theorem prover. In *Proc. of ACM SAC'08*, pages 333–338, Brazil, 2008. ACM Press.

[3] 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.

[4] H. Amjad. Programming a symbolic model checker in a fully expansive theorem prover. In *Proceedings of TPHOLs'03*, volume 2758 of *LNCS*, pages 171–187. Springer-Verlag, 2003.

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

[6] Cadence Design Systems. V2.3. *FormalCheck Users Guide*, August 1999.

[7] W. Clocksin and C. Mellish. *Programming in Prolog*. Springer Verlag, 1987.

[8] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 439–448, New York, NY, USA, 2000. ACM.

[9] 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.

[10] J. Filliâtre and C. Marché. The why/krakatoa/caduceus platform for deductive program verification. In *CAV*, pages 173–177, 2007.

[11] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

[12] M. J. C. Gordon. Reachability programming in HOL98 using BDDs. In *International Conference on Theorem Proving in Higher Order Logics TPHOLs*, Lecture Notes in Computer Science, pages 179–196, 2000.

[13] M. J. C. Gordon. Programming combinations of deduction and BDD-based symbolic calculation. *LMS Journal of Computation and Mathematics*, 5:56–76, August 2002.

[14] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, New York, NY, USA, 1993.

[15] John Harrison. Binary decision diagrams as a HOL derived rule. *The Computer Journal*, 38:162–170, 1995.

[16] 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.

[17] T. Kropf. *Introduction to Formal Hardware Verification*. Springer Verlag, 1999.

[18] M. Krykhtin, Y. Mokhtari, O. Ait Mohamed, and X. Song. Towards software model checking using MDGs. *The 2nd Annual IEEE Northeast Workshop on Circuits and Systems, 2004. NEWCAS 2004.*, pages 345–348, June 2004.

[19] T. Mhamdi and S. Tahar. Providing automated verification in HOL using MDGs. In *Automated Technology for Verification and Analysis*, pages 278–293, 2004.

[20] Network Processing Forum. *Look-Aside (LA-1) Interface, Implementation Agreement, Revision 1.1*. Kluwer Academic Publishers, April 15, 2004.

[21] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. of the 15th IEEE International Conference on Automated Software Engineering*, 2000.

[22] 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.

[23] Z. Zhou and N. Boulerice. *MDGs Tools (V1.0) User's Manual*. D'IRO, University of Montreal, June 1996.

[24] Z. Zhou, X. Song, S. Tahar, E. Cerny, F. Corella, and M. Langevin. Formal verification of the island tunnel controller using multiway decision graphs. In *Proc. of FMCAD '96*, pages 233–247, London, UK, 1996. Springer-Verlag.