The Verification of MDG Algorithms in the HOL Theorem Prover

Sa'ed Rasmi H. Abed

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy at

Concordia University

Montréal, Québec, Canada

June 2008

Your file Votre référence
ISBN: 978-0-494-42548-0
Our file Notre référence
ISBN: 978-0-494-42548-0

ABSTRACT

The Verification of MDG Algorithms in the HOL Theorem Prover

Sa'ed Rasmi H. Abed, Ph. D.

Concordia University, 2008

Formal verification of digital systems is achieved, today, using one of two main approaches: states exploration (mainly model checking and equivalence checking) or deductive reasoning (theorem proving). Indeed, the combination of the two approaches, states exploration and deductive reasoning promises to overcome the limitation and to enhance the capabilities of each. Our research is motivated by this goal. In this thesis, we provide the entire necessary infrastructure (data structure + algorithms) to define high level states exploration in the HOL theorem prover named as MDG-HOL platform. While related work has tackled the same problem by representing primitive Binary Decision Diagram (BDD) operations as inference rules added to the core of the theorem prover, we have based our approach on the Multiway Decision Graphs (MDGs). MDG generalizes ROBDD to represent and manipulate a subset of first-order logic formulae. 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 function. Considering MDGs instead of BDDs will raise the abstraction level of what can be verified using a state exploration within a theorem prover. The MDGs embedding is based on the logical formulation of an MDG as a Directed Formulae (DF). The DF syntax is defined as HOL built-in data types. We formalize the basic MDG operations using this syntax within HOL following a deep embedding approach. Such approach ensures the consistency of our embedding. Then, we derive the correctness

proof for each MDG basic operator.

Based on this platform, the MDG reachability analysis is defined in HOL as a conversion that uses the MDG theory within HOL. Then, we demonstrate the effectiveness of our platform by considering four case studies. Our obtained results show that this verification framework offers a considerable gain in terms of automation without sacrificing CPU time and memory usage compared to automatic model checker tools.

Finally, we propose a reduction technique to improve MDGs model checking based on the MDG-HOL platform. The idea is to prune the transition relation of the circuits using pre-proved theorems and lemmas from the specification given at system level. We also use the consistency of the specifications to verify if the reduced model is faithful to the original one. We provide two case studies, the first one is the reduction using SAT-MDG of an Island Tunnel Controller and the second one is the MDG-HOL assume-guarantee reduction of the Look-Aside Interface. The obtained results of our approach offers a considerable gain in terms of heuristics and reduction techniques correctness as to commercial model checking; however a small penalty is paid in terms of CPU time and memory usage.

To My Family

ACKNOWLEDGEMENTS

I would like to express my gratitude to my supervisor, Dr. Otmane Ait Mohamed, whose expertise, understanding, and patience, added considerably to my graduate experience. I am deeply grateful for his support and encouragement throughout my Ph.D. studies.

I would like to thank the other members of my committee, Dr. Sofiène Tahar, Dr. Rachida Dssouli, and Dr. Asim J. Al-Khalili for the assistance they provided at all levels of the research project. Finally, I would like to thank Dr. El-Mostapha Aboulhamid from Montreal University for taking time out from his busy schedule to serve as my external examiner.

Very special thanks go out to my colleagues in the Hardware Verification Group (HVG), without their help, motivation and encouragement I would not have reached this point in my research. I have spent three years and half in the HVG labs and will never forget the great moments, and achievements we had together during these years. Also, I would like to thank Dr. Ghiath Al Sammane, a post doctoral in our (HVG) group, for many discussions and helpful suggestions, which are invaluable to this thesis.

Last but not least, I would like to reserve my deepest thanks for my parents, sisters and brother, for their support and encouragement. My wife, who has been with me in every moment of my PhD tenure, is my source of strength and without her support this thesis would never have started much less finished. I would like to mention my children, Bara', Baha and Lina, for bringing joy and fun in my life and for their sacrifices and patience. I can never thank them enough.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

CAD	Computer Aided Design
ASM	Abstract State Machine
BDD	Binary Decision Diagram
ROBDD	Reduced Ordered Binary Decision Diagram
SAT	Satisfiability Checking
DF	Directed Formulae
DAG	Directed Acyclic Graph
FSM	Finite State Machine
HDL	Hardware Description Language
MDG	Multiway Decision Graph
MC	Model Checking
BMC	Bounded Model Checking
STE	Symbolic Trajectory Evaluation
LTL	Linear time Temporal Logic
CTL	Computational Tree Logic
HOL	Higher-Order Logic
ATP	Automatic Theorem Prover
LCF	Logic of Computable Function
ML	Meta Language
FL	Functional Language
CNF	Conjunctive Normal Form
QFB	Quantified Boolean Formulae
RTL	Register Transfer Level

LHS	Left Hand Side
RHS	Right Hand Side
VLSI	Very Large Scale Integration
ITC	Island Tunnel Controller
LA-1	Look-Aside Interface
RelP	Relational Product
PbyS	Pruning by Subsumption

Chapter 1

Introduction

With the increasing complexity of the design of digital systems and the size of the circuits in VLSI technology, the role of design verification has gained a lot of importance. Serious design errors and bugs take a lot of time and effort to be detected and corrected especially when they are discovered late in the verification process. This will increase the total cost of the chip. In order to overcome these limitations, *formal verification techniques* arose as a complement to simulation for detecting errors as early as possible, thus ensuring the correctness of the design.

Formal techniques are the application of applied mathematics in order to prove that the design implementation satisfies its specifications, and entail reasoning in some formal logic. In general, *formal verification* of digital systems is achieved, today, using one of two main approaches: states exploration [49] (mainly model checking and equivalence checking) or deductive reasoning (theorem proving). It is accepted that both approaches have complementary strengths and weaknesses.

State exploration approaches use states space traversal algorithms on finite-state models to check if the implementation satisfies its specification. They are focused mostly on automatic decision procedures for solving the verification problem. In case

the verification fails, the user can track with the counter example produced as to why it failed.

Model checking is an automatic approach for verifying finite-state systems and mainly used in hardware and protocol verification. The circuit is described as a state machine with a transition to describe the circuit behavior. The specifications are described as properties that the machine should satisfy. Furthermore, they can produce a counterexample when the property does not hold, which can be very important for correcting the corresponding error in the implementation under verification or in the specification itself. Traditionally, model checkers used explicit representations of the state transition graph, for all but the smallest state machines. However, model checking suffers from the state explosion problem [19]: the number of the explored states grows exponentially in the size of the system description.

Equivalence checking is used to prove functional equivalence of two design representation modeled at the same or different levels of abstraction. It can be divided into combinational equivalence checking and sequential equivalence checking. Combinational equivalence checking is based on the canonical representations of Boolean functions or Binary Decision Diagrams (BDDs). Equivalence checking verifies for all input sequences that an implementation has the same outputs as the specification, both modeled as Finite State Machines (FSM). Sequential equivalence checking is used to verify the equivalence between two sequential designs at each state. Sequential equivalence checking consider only the behavior of two designs while ignoring their implementation details such as register mapping. It can verify the equivalence between Register Transfer Level (RTL) and netlist or RTL and the behavioral model which is very important in design verification. The disadvantage of sequential equivalence checking is that it cannot handle large design because it enumerates state space

explosion problem very fast.

In deductive reasoning approach, the correctness of a design is formulated as a theorem in a mathematical logic and the proof is checked using a general-purpose theorem-prover. Based on first-order and high-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 the state explosion problem when used to reason formally about such data and relationships. Unfortunately, if the property fails to hold, deductive methods do not give a counterexample. Furthermore, this frequent situation requires skilled manual guidance for verification and human insight for debugging. Yet theorem provers, today, provide feedback, but only expert user can track the proof trace and determine whether the fault lies within the system, the property being verified, or within the failed proof tactic.

There has been a great deal of work over the past decade to combine the two approaches to gain the strengths of both, and alleviate the weaknesses. Successful combinations of this kind have been achieved in [2, 44, 46, 48, 57, 75, 78]. The strengths and weaknesses of model checking and deductive theorem proving, as discussed above, are summarized in Table 1.1.

Table 1.1: Deductive theorem proving vs. state exploration method

Method	State exploration method	Deductive method	Hybrid method
Automation	completely automatic	interactive	semi-automatic
Domain size	finite system (large)	infinite system (complex)	finite system (very large)
Debugging	generates counter-example	expert based	rarely generates counter-example

The combination of the two approaches can be performed either by adding a

layer of deduction theorems and rules on top of the model checking tool (hybrid approach) or by embedding model checking algorithms inside theorem provers (deep embedding approach). Our research is motivated by using the deep embedding approach to blend the best of model checker and theorem prover.

The structure of the rest of this chapter is as follows: In Section 1.1, we briefly introduce the formal verification techniques. Section 1.2 surveys the literature and presents the related work. An overview of the research and the contribution of this thesis is explained in Sections 1.3 and 1.4, respectively. Finally, the outline of the thesis is presented in Section 1.5.

1.1 Formal Verification Techniques

Formal verification problem consists of mathematically establishing that an implementation behaves according to a given set of requirements or specification. To classify the various approaches, we first look at the three main aspects of the verification process: the system under investigation (implementation), the set of requirements to obey (specification) and the formal verification tool to verify the process (relationship between implementation and specification).

The implementation refers to the description of the design that is to be verified. It can be described at different levels of abstraction which results in different verification methods. Another important issue with the implementation is the class of the system or circuit to be verified, i.e., whether it is combinational/sequential, synchronous/asynchronous, pipelined or parameterized hardware. These variations may require different approaches. The specification refers to the property with respect to which the correctness is to be determined. In practice, one needs to model both the implementation and the specification in the tool, and then uses one of the formal

verification algorithms of the tool to check the correctness of the system or in some cases give a trace of error (counter-example).

Formal techniques have long been developed within the computing research community as they provide sound mathematical foundation for the specification, implementation and verification of computer system. Thus, formal verification is proposed as a method to help certify hardware and software, and consequently, to increase confidence in new designs. A correctness proof cannot guarantee that the real device will never malfunction; the design of the device may be proved correct, but the hardware actually built can still behave in a way unintended by the designer. Wrong specification can play a major rule in this, because it has been verified that the system will function as specified, but it has not been verified that it will work correctly. Defects in physical fabrication can cause this problem too. In formal verification a model of the design is verified, not the real physical implementation. Therefore, a fault in the modeling process can give false negatives (errors in the design which do not exist). Although sometimes, the fault covers some real errors.

Formal verification approaches can be generally divided into two main categories: theorem proving methods and state exploration methods such as model checkers as described in the following subsections.

1.1.1 Theorem Proving

Theorem proving is an approach where the specification and the implementation are usually expressed in first-order or higher-order logic. Their relationship is formed as a theorem to be proved within the logic system. This logic is a set of axioms and a set of inference rules. Steps in the proof appeal to the axioms and rules, and possibly derived definitions and intermediate lemmas. The axioms are usually "elementary"

in the sense that they capture the basic properties of the logic's operators [32].

Theorem proving utilizes the proof inference technique. The problem itself is transformed into a sequent, a working representation for the theorem proving problem. Then a sequent holds if the formula f holds in any model:

$$\models f$$

A proof system is collection of *inference rules* of the form:

$$\texttt{(name)}\ \frac{P_1 \ldots P_n}{C}$$

where C is a conclusion sequent, and P_i's are premisses sequents. The meaning of an inference rule is, if all the premisses are derivable, then the conclusion is guaranteed to hold. Some inference rules may have no premisses, in which case their conclusion automatically holds. Such rules are also called *axioms*, and they are the only means to complete the proof derivation.

Traditionally, the logic used in theorem proving is the classical First- or Higher-Order logic (FOL and HOL respectively). Some other kinds of logics are also used, but since all of them can be expressed in the higher-order logic, the latter is used much more often as a general property language.

Theorem proving methods have been in use in hardware and software verification for a number of years in various research projects. Some of the well-known theorem provers are HOL (Higher-Order Logic), ISABELLE, PVS (Prototype Verification System), Coq and ACL2 [23, 36, 42, 47, 66]. These systems are distinguished by, among other aspects, the underlying mathematical logic, the way automatic decision procedures are integrated into the system, and the user interface. Even though they are powerful, they require expertise in using a theorem prover. User is expected to know the whole design leading to a white box verification approach. It is not fully

automated and requires a large amount of time to verify the system. Another shortcoming is the inability to produce counter-examples in the event of a failed proof, because the user does not know whether the required property is not derivable or whether the person conducting the derivation is not ingenious enough. The advantage of the deductive verification approach is that it can handle very complex systems because the logics of theorem provers are more expressive. In the next chapter, we will overview the HOL theorem proving system, which we intend to use in this thesis.

1.1.2 Model Checking

Model checking is a state exploration based verification technique developed in the 1980s by Clarke and Emerson [19] and independently by Quielle and Sifakis [68]. In model checking, a state of the system under consideration is a snapshot of the system at certain time, given by the set of the variables values of that system at that time. The system is then modeled as a set of states together with a set of transitions between states that describe how the system moves from one state to another in response to internal or external stimulus. Model checking tools are then used to verify that desired properties (expressed in some temporal logic) hold in the system.

Model checker has two important advantages. First, once the correct design of the system and the required properties has been fed in, the verification process is fully automatic. Second, in the event of a property not holding, the verification process is able to produce a counter-example (i.e. an instance of the behavior of the system that violates the property) which is extremely useful in helping the human designers pinpoint and fix the flaw. On the other hand, model checkers are unable to handle very large designs due to the state space explosion problem [19]. Another drawback is the problematic description of specifications as properties, this description sometimes

may not give full system coverage.

Model checkers such as SPIN [40], COSPAN [51], SMV [54], and MDG [88] take as input, essentially, a finite-state system and temporal property in some variety or subset of Computation Tree Logic (CTL*), and automatically check that the system satisfies the property. Moreover, the model is often restricted to a finite-state transition system, for which finite-state model checking is known to be decidable. The design or model is formalized in terms of a state machine (*Transition System*), or a Kripke structure:

$$M = (P, S, I, R, L)$$

where M is a state machine (model) with a transition to describe the circuit behavior, P is a set of atomic propositions, S is a finite set of states, $I \subseteq S$ is a set of initial states, $R \subseteq SxS$ is a transition relation that must be total (i.e. for every $s \in S$ there exists $s' \in S$ such that $(s' \in R)$), and $L : S \rightarrow 2^P$ maps each state to the set of atomic propositions true in that state. The property ϕ is formalized as a logical formula that the machine should satisfy. The verification problem is stated as checking the formula ϕ in the model M:

$$M \vDash \phi$$

If the model M is represented explicitly as a transition relation, then the size of the model is limited to the number of states that can be stored in the computer memory, which are a few million states with the current technology. To increase the size of the model, more efficient state representations can be used to manipulate these formulae using BDDs or SAT solving techniques.

Binary Decision Diagrams

Binary Decision Diagrams (BDDs) [13] are data structures used as a compact representation for the Boolean function which improves the capacity of the model checker. BDDs have several useful properties. Many common functions have small BDDs in addition to the fact that the BDDs are easy to manipulate. Also a function can be evaluated in linear time in the number of variables and also can be existentially or universally quantified (Boolean) variables in time quadratic in the size of the BDD. Moreover, the order in which the variables appear can be fixed and hence the BDD is a canonical representation for the Boolean function.

BDDs are used to overcome the capacity limitation of the model checkers, different representations of ROBDDs (Reduced Order Binary Decision Diagrams) [14] are used to manipulate the state transition relations as diagrams and this allows model checkers to verify larger systems. Still, most model checkers face the state space explosion problems [19] even using Symbolic Model Checking. To be able to apply model checking to larger designs, state reduction techniques are used that exploit some features of the model, the properties, or the problem domain to reduce the state space to a tractable size. Examples include partitioned transition relation, dynamic variable reordering, cone of influence reduction, abstraction, problem-specific techniques, e.g. when the original design is rewritten in a simpler way, omitting the irrelevant details, but preserving the important behavior for the property being verified.

In this thesis, we intend to use the Multiway Decision Graphs (MDGs), a new class of decision graph. MDG was proposed as a solution to the state space explosion problem [21]. In MDGs based model checking approach, data signals are denoted by abstract variables, and data operators are represented by uninterpreted function symbols. As a result, a verification based on abstract-implicit-state-enumeration can

be carried out independently of datapath width, substantially lessening the state explosion problem. Table 1.2 shows the abstraction level of MDG corresponding to traditional methods.

Table 1.2: Raising the Abstraction Level

Conventional Method	**Multiway Decision Graphs**
ROBDD [14]	MDGs [21]
Finite State Machine (FSM)	Abstract State Machine (ASM)
Implicit state enumeration	Abstract state implicit enumeration of ASM
CTL based model-checking	Based on first-order abstract CTL*

SAT Based Methods

An alternative for decision graphs is to represent the transition relation in CNF and use Satisfiability Checking (SAT) [26, 81] with several properties that make them attractive compared to BDDs. SAT solvers can decide satisfiability of very large Boolean formulae in reasonable time, but they are not canonical and require additional efforts to check for equivalence of formulas. As a result, various researchers have developed routines for performing Bounded Model Checking (BMC) [3, 11, 30] using SAT. The common theme is to convert the problem of interest into a SAT problem, by devising the appropriate propositional Boolean formula, and to utilize other non-canonical representations of state sets. However, they all exploit the known ability of SAT solvers to find a single satisfying solution when it exists. Moreover, SAT solver technology has improved significantly in recent years with a number of sophisticated packages now available. Well known state-of-the-art SAT solvers include CHAFF [59], GRASP [52] and SATO [89]. Since state sets can be represented as Boolean formulae,

and since most model checking techniques manipulate state sets, SAT solvers have enormously boosted their speed and applicability.

1.2 Related Work

Model checking is automatic while theorem proving is not. On the other hand, theorem proving can handle complex systems while model checking can not. Today, there exist a number of integration tools of theorem proving and model checking. The motivation is to achieve the benefits of both tools and to make the verification simpler and more effective. In this section, we explore two approaches of linking proof systems to external automated verification tools. The approaches can be divided in two kinds:

1. Hybrid approach: adding a layer of deduction theorems and rules on top of Decision Diagrams tool, i.e. combining theorem provers with other powerful model checking tool.

2. Deep embedding approach: adding Decision Diagrams algorithms to theorem provers.

We first review the most related work to every approach and then, we contrast between them according to their efficiency, complexity and feasibility.

1.2.1 Hybrid Approach

The hybrid approach implements a tool linking model checking and theorem proving. During the verification procedure, the user deals mainly with the theorem proving tool. Verification using hybrid approach proceeds as shown in Figure 1.1. The user starts by providing the theorem proving with the design (specification or implementation), the property and the goal to be proven. If the goal fits the required pattern, the

theorem proving tool generates the required model checking files (sub-goals). The latter are sent to the model checking tool for verification. If the property holds, a theorem is created (Make-Theorem). Otherwise, the proof is performed interactively.

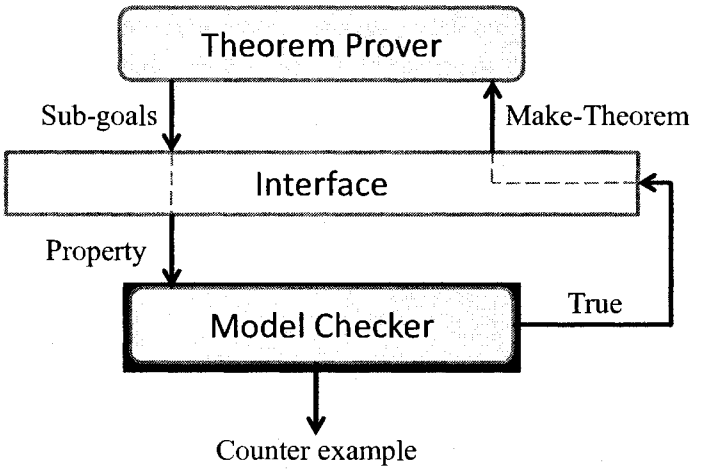


Figure 1.1: Theorem Proving and Model Checking Interface

The linkage between both tools is carried out using scripting languages (translators) to be able to automatically verify small subgoals generated by the theorem prover from a large system. The disadvantage of this approach lies in achieving an efficient and correct translation from theorem prover logic to a model checker and from model checker to theorem prover (import the result or give a counter-example). Successful combinations of this kind have been achieved in [46, 48, 57, 75, 78].

Rajan et al. [74, 75] described an approach where a BDD-based model checker for the propositional μ-calculus has been used as a decision procedure within the framework of PVS. An extension of the μ-calculus, which consists of Quantified Boolean Formulae (QFB), is defined using PVS higher-order logic. The temporal operators are then defined using the μ-calculus. These temporal operators apply to

arbitrary state spaces. In the case where the state type is constructed in a hereditarily finite manner, μ-calculus expressions are translated into input acceptable by a μ-calculus model checker. This model checker can then be used as a decision procedure to prove certain subgoals. The model checker accepts the translated input from μ-calculus expression. The generated subgoals are verified by the model checker and the results are used in the proof process of PVS.

Schneider et al. [46] used higher order hardware formulae to express the safety and liveness properties hierarchically. They proposed an approach of invoking model checking within HOL where properties are translated from HOL to temporal logic. A new class of higher-order formulae was presented, which allows a unified description of hardware structure and behavior at different levels of abstraction. Datapath oriented verification goals involving abstract data types can be expressed by these formula as well as control dominated verification goals with irregular structure. To ease the proofs of the goals in HOL, a translation procedure was presented which converts the goals into several Computational Tree Logic (CTL) model checking problems, which are then solved outside HOL.

Schneider and Hoffmann [78] linked the SMV model checker to HOL using PROSPER. It provides an open proof architecture for the integration of different verification tools in a uniform higher-order logic environment. They embedded the linear time temporal logic (LTL) in HOL and translated LTL formulae into ω-Automata, a form that can be reasoned about within SMV. The translation is completely implemented by means of HOL rules. HOL terms are exported to SMV through the PROSPER plug-in interface. On successful model checking, the results are returned to HOL and turned to theorems. This integration tool allows SMV to be used as a HOL decision procedure. The deep embedding of the SMV specification language in

HOL allows LTL specifications to be manipulated in HOL.

In [67], Pisini and Tahar proposed a hybrid approach for formal hardware verification which uses the strengths of the HOL theorem prover and the advantages of the automated tool MDG which supports equivalence and model checking. They developed a linkage tool between HOL and MDG which uses the specification and implementation of a circuit written in HOL to automatically generates all required MDG files. The implementation of the methodology is achieved by building a linkage tool using Moscow ML to translate from HOL to MDG. It then calls the MDG equivalence checking procedure and reports the MDG verification results back to HOL.

The MDG-HOL system [48] is a hybrid system which links the HOL interactive proof system and the MDG automated hardware verification system. It supports a hierarchical verification approach and fits the use of MDG verification naturally within the HOL framework for a compositional hierarchical verification. The HOL system is used to manage the proof. The MDG system is called to verify the submodules of a design. When the MDG-HOL system is used to verify a design, the design is modeled as a hierarchy structure with modules divided into submodules.

An extension of the above work was presented in [57] to link HOL and the MDG model checker. They described a hybrid tool that links the HOL theorem prover and the MDG model checker. For this purpose, they developed an interface which reads a HOL goal, generates the required MDG files, calls the MDG model checker, and generates the HOL theorem on successful verification. The interface between the two tools is implemented using ML.

1.2.2 Deep Embedding Approach

In this approach, the emphasis is to establish a secure platform for new verification algorithms. The performance penalty will be compensated by the secure infrastructure. The approach implements a model checking inside a theorem proving tool. As shown in Figure 1.2, the design and the property are fed to the model checking to check if the property holds and create a theorem. Otherwise, the proof cannot be performed.

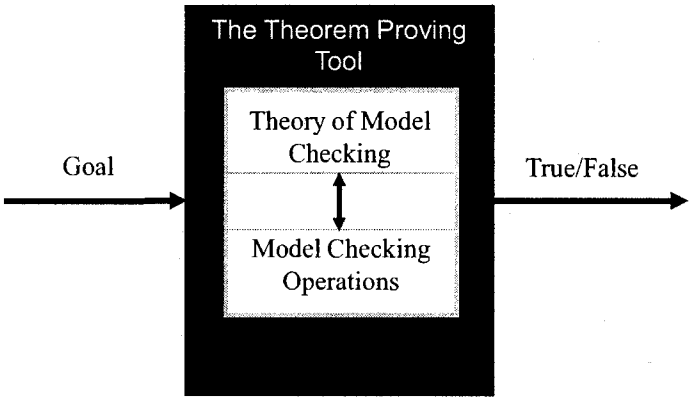


Figure 1.2: Embedding Model Checking inside Theorem Proving Tool

The result of the model checker is correct by construction, since both of the theory and the implementation are proved correct in the theorem prover. Thus a high assurance of soundness is guaranteed because more work is backed up by mechanized fully-expansive proof. The price for the extra proof and flexibility is in increased development effort. This approach differs from the hybrid approach in the way the verification is performed. In fact, we do not use an external checking tool, instead we deeply embed the model checker algorithms inside the theorem prover. Thus the criteria of correctness by construction, efficiency, flexibility and expressiveness can be met. Successful works have been achieved in [7, 34, 35, 37, 44, 56].

The "deep embedding" approach [69] introduce the model checker syntax as a

new higher order logic type and then define the operations and algorithms based on this syntax within the theorem prover. This contrasts within a "shallow embedding" where the syntax is not formally represented in the logic, only in the meta-language. In general, a deep embedding allows one to reason about the language itself rather than just the semantics of programs in the language.

We consider two categories of related work: the first category regarding embedding of model checking algorithms in theorem provers. The second category regarding correctness proof of the model checker algorithms.

Embedding of Model Checking Algorithms in Theorem Provers

Model checkers [54] are usually built on top of BDDs [13], or some other set of efficiently implemented algorithms for representing and manipulating Boolean formulae. The closest work, in approach to our own is that of Joyce and Seger [79], Gordon [34, 35] and later Amjad [7].

Voss system [79], an implementation of Symbolic Trajectory Evaluation (STE), was implemented in a lazy Functional Language (FL). In [44] Voss was interfaced to HOL and the verification using a combination of deduction and STE was demonstrated. The HOL-Voss system integrates HOL88 deduction with BDD computations. The BDD tools are programmed in FL as a built-in datatype. The assertion language of Voss was formalized in HOL and a HOL tactic, which can make an external calls to the Voss system, checks wether an assertion is true. Then the proved assertion was returned as a HOL theorem. The early experiments with HOL-Voss suggested that a lighter theorem prover component was sufficient, since all that was needed is a way to combine results obtained from STE. A system based on this idea, called Voss-ThmTac, was later developed by Aagaard *et al.* [2]; combination of the ThmTac

theorem prover with the Voss system. Then the development of HOL-Voss evolved into a new system called Forte [1]. More recently, with industrial take-up at Intel, Forte [55] has become one of the most mature formal verification environments based on tool integration.

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 [35].

In [37], Harrison implemented BDDs inside the HOL system without making use of external oracle. The BDD algorithms were used by a tautology-checker. However, the performance was about thousand times slower than with a BDD engine implemented in C. Harrison argued that by re-implementing some of HOL's primitive rules, the performance could be improved by around ten times.

Amjad [7] demonstrated how BDD based symbolic model checking algorithms for the propositional *μ-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 [35] 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. Thus

the security of the theorem prover is compromised only to the extent that the BDD engine or the BDD inference rules may be unsound.

Our work, deals with embedding MDGs [21] rather than BDDs. In fact, while BDDs are widely used in state-exploration methods, they can only represent Boolean formulae. By contrast, MDGs represent a subset of first-order terms allowing the abstract representation of data and hence raising the level of abstraction. Another major difference is that it implements the related inference rules for BDD operators in the core of HOL as a plugged in code, whereas we implement the MDG operations inside HOL itself.

Mhamdi and Tahar [56] follow a similar approach to the BuDDy work [35]. The work builds on the MDG-HOL [48] project, but uses a tightly integrated system with the MDG primitives written in ML rather than two tools communicating as in MDG-HOL system. In their work, 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.

Correctness Proof of Model Checking Algorithms

Verification of BDD algorithms has been a subject of active research using different proof assistants such that HOL, PVS, Coq, and ACL2 [23, 36, 42, 47]. A common goal of these papers is to extend the prover with a certified BDD package to enhance the BDD performance, while still inside a formal proof system. Moreover, there is a general consensus in the formal verification community that correctness proofs should be checked, partly or wholly, by computers. Some efforts have been made to verify model checkers and theorem provers.

In [71], the authors successfully carried out the verification task of the RAVEN model checker. RAVEN is a real-time model checker which uses time-extended finite state machines (interval structure) to represent the system and a timed version of CTL (CCTL) to describe its properties. The specification and the correctness proof were carried out using an interactive specification and verification system KIV.

In [62], the author showed a mechanism of how certifying model checker can be constructed. The idea is that, a model checker can produce a deductive proof on either success or failure. The proof acts as a certificate of the result, since it can be checked independently. A certifying model checker thus provides a bridge from the model-theoretic to the proof-theoretic approach to verification. The author developed a deductive proof system for verifying branching time properties expressed in the μ-calculus, and showed it to be sound and relatively complete. Then, a proof generation in this system from a model checking run is presented. This is done by storing and analyzing sets of states that are generated by the fixpoint computations performed during model checking.

Krstic and Matthews [50] provided a technique for proving correctness of high performance BDD packages. In their work, they adopted an abstraction method called monadic interpretation for verifying an abstraction of the BDD programs with the primitives specified axiomatically. The method is suitable for higher order logic theorem provers such as Isabelle/HOL. The monadic interpreter translates source programs of input type A and output type B into function of type $A \Rightarrow MB$ in the target functional language, where the type constructors M is a suitable monad that encapsulate the notion of computation used by the source language to describe BDD programs. At this level, they modeled the BDD programs as a function in higher order logic in the style of monadic interpreters. Then the correctness proof was carried out

on the BDD abstract model.

Wright [86] described an embedding of higher order proof theory within the logic of the HOL theorem proving system. Types, terms and inferences were represented as new types in the logic of the HOL system, and notions of proof and provability were defined. Using this formalization, it was possible to reason about the correctness of derived inference rules and about the relations between different notions of proofs: a Boolean term is provable if and only if there exists a proof for it. The formalization is also intended him to make it possible to reason about programs that handle proofs as their data (e.g., proof checker).

Harrison [38] answered a question concerning the correctness of the theorem prover itself. The author verified formally that the abstract HOL logic is correct and that the OCaml code does correctly implement this logic. The verification is conducted with respect to a set-theoretic semantics within the HOL Light itself.

The authors in [85] implemented and proved the correctness of BDD algorithms using Coq. One of their goals was to extract a certified algorithm manipulating BDDs in Caml (the implementation language of Coq). BDDs were represented as DAGs and maps were used to model a state of the memory in which all the BDDs are stored. The authors used reflection to prove a given property P applied to some term t where the program is described and proved in Coq. This means that writing a program π that takes t as an input and returns true exactly when $P(t)$ holds. Then, to show π is correct with respect to P they needed to be sure that whenever $\pi(t)$ returns true $P(t)$ holds and this is done inside the Coq proof assistant itself (i.e. the proof of P has been replaced by the computation of π and reflect this by allowing the system to accept meta-level computation as actual proof).

Another concept to prove the program correctness using Hoare logic as described

by Ortner and Schirmer [64]. The principle of this logic is to annotate the program with pre- and post-conditions and to observe the changes made by each statement of the program. Ortner and Schirmer modeled the graph structure of the BDD as a kind of heap and presented the verification of BDD normalization. They follow the original algorithm presented by Bryant in [13]: transforming an ordered BDD into a reduced, ordered and shared BDD. The work is based on Schirmer's research on the Verification Condition Generator (VCG) to generate the proof obligations for Hoare Logic. The proofs are carried out in the theorem prover Isabelle/HOL.

Haiyan et al. [87] verified formally the linkage between a simplified version of MDG tool and the HOL theorem prover. The verification is based on the importing of MDG results to HOL theorems. Then, they combined translator correctness theorems with the linkage theorems in order to allow low level MDG verification results to be imported into HOL in terms of the semantics of MDG-HDL. The work was concerned with ways of increasing trust in the linked systems.

Our work follows the verification of the Boolean manipulating package, but using MDG instead. We provided a complete formalization of the MDG logic and its well-formedness conditions as DFs in HOL mechanically. Based on this infrastructure we formalized the basic MDG operations in HOL following a deep embedding approach and proved their correctness. Our work focuses more on how one can raise the level of assurance by embedding and proving formally the correctness of those operators in HOL to use them as an infrastructure for MDG model checker.

1.3 Proposed Methodology

The intention of our research is to provide a secure platform that combines an automatic high level MDGs model checking tool within the HOL theorem prover. While

related work has tackled the same problem by representing primitive Binary Decision Diagrams (BDD) operations [13] as inference rules added to the core of the theorem prover [35], we have based our approach on the Multiway Decision Graphs (MDGs) [21]. 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 rise the abstraction level of what can be verified using a state exploration within a theorem prover. Furthermore, an MDG structure in HOL allows better proof automation for larger datapaths systems.

In this thesis, we provide the entire necessary infrastructure (data structure + algorithms) to define a high level state exploration in the HOL theorem prover named as MDG-HOL platform.

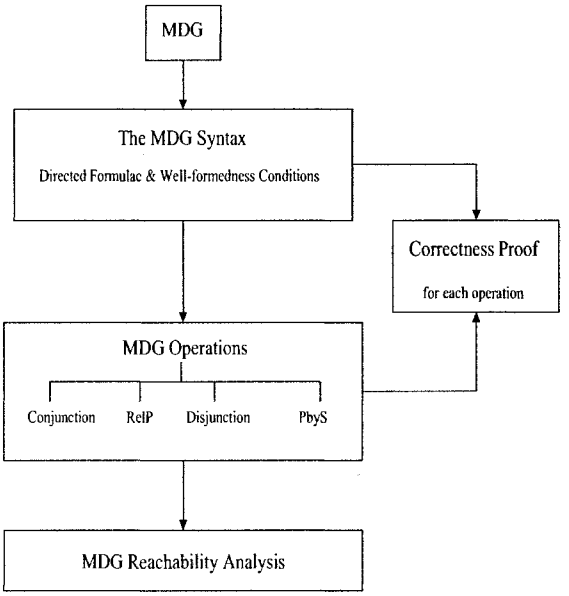


Figure 1.3: Overview of the Embedding Methodology in HOL

Firstly, as shown in Figure 1.3, we define the MDG structure inside the HOL theorem prover to be able to construct and manipulate MDGs as formulae in HOL. This step implies a formal logic representation for the *MDG Syntax*. This representation is based on the Directed Formulae DF: an alternative vision for MDG in terms of logic and set theory [6]. Secondly, a HOL tactic is defined to check the satisfaction of the well-formedness conditions of any directed formula.This step is important to guarantee the canonical representation of the MDG as a DF. Then, the definition of each MDG operations is defined and a correctness proof is derived within HOL.

Based on this platform, the MDG reachability analysis is defined in HOL as a conversion that uses the MDG theory within HOL.Then, we demonstrate the effectiveness of our platform by considering four case studies.Our obtained results show that this verification framework offers a considerable gain in terms of automation without sacrificing CPU time and memory usage compared to automatic model checker tools.

Finally, we propose a reduction technique to improve MDGs model checking based on the MDG-HOL platform. The idea is to prune the transition relation of the circuits using pre-proved theorems and lemmas from the specification given at system level. We also use the consistency of the specifications to verify if the reduced model is faithful to the original one. We provide two case studies, the first one is the reduction using SAT-MDG of an Island Tunnel Controllerand the second one is the MDG-HOL assume-guarantee reduction of the Look-Aside Interface. The performance penalty in the case of SAT-MDG reduction verification is acceptable as compared with commercial model checking tools. In the case of assume guarantee in MDG-HOL, the reduction strategy results still satisfactory in terms of heuristics and reduction techniques correctness, however a small penalty is paid in terms of time and memory.

1.4 Thesis Contributions

The objective of our research is to explore a way of increasing the degree of trust of the MDG system by embedding the MDG system in HOL. In light of the above related work review, proposed methodology, and discussions, we believe the contributions of the thesis can be specified as follows:

- We have provided a secure platform (data structure + algorithms) of MDG system in HOL. This step consists of the following phases:

  1. Embedding of the MDG formal logic underlying the abstract state machines in HOL.
  2. Defining the notion of well formed HOL terms. These terms could be represented canonically by MDGs.
  3. Embedding the MDG algorithms (conjunction, relational product (RelP), disjunction, and pruning by subsumption (PbyS)) following deep embedding approach. Also, we have two kinds of theorems: one theorem regarding the correctness proof of each MDG operation, and the other one for preserving the well-formedness of the operation results.

- The MDG based reachability analysis is then defined in HOL as a conversion that uses the MDG-HOL platform and a fixpoint theorem is then proven for some particular circuits.

- We have evaluated the performance of the MDG-HOL platform using a set of benchmarks to ensure the applicability of our approach.

- We have proposed a reduction methodology to improve the MDGs model checking as well as to verify the soundness of model checking reduction techniques.

- We have provided two case studies, the first one is the reduction using the SAT-MDG technique of the Island Tunnel Controller (ITC), and the second one is the MDG-HOL assume-guarantee reduction technique of the Look-Aside Interface (LA-1).

In summary, we have created a new formal theory for MDGs (data structure + operations) inside the HOL theorem prover which provides us with several theoretical advantages without too high performance penalty. We used this theory or platform to verify the soundness of model checking reduction techniques. We thus hope that this work will be of interest to the research community and also be of use to industrial practitioners.

1.5 Thesis Organization

The rest of the thesis is organized as follows:

- In Chapter 2, we review the basics of the HOL theorem prover. We also introduce the basic concepts of Multiway Decision Graphs (MDGs).
- Chapter 3 presents the formal logic underlying MDGs as well as well-formedness conditions and its embedding inside HOL.
- In Chapter 4, we formalize the MDG basic operations and prove the correctness of each operation.
- In Chapter 5 , we show the formalization of the MDG reachability analysis and the proposed conversion for proving a fixpoint. We also consider four case studies to measure the performance of the MDG-HOL platform.

- Chapter 6 considers the applications and case studies for the proposed reduction techniques.
- Chapter 7 concludes the thesis and indicates the future work.

Chapter 2

Preliminaries

In this chapter, we give a brief description to the HOL theorem prover as well as to the Multiway Decision Graphs (MDGs) system. The intent is to familiarize the reader with the main concepts and notations that are used in the rest of the thesis. Section 2.1 starts by a basic description of higher-order logic concepts. Then, we describe the syntax and semantics of the particular logical system supported by HOL notation, as well as the proof methods supported by the HOL theorem prover.

Section 2.2 describes the underlying formal logic of MDGs, the MDGs structure, the Abstract State Machine (ASM), the MDG tool and the MDG model checker.

2.1 The HOL Theorem Prover

The HOL system is an LCF [33] (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 and

algorithms verification, modeling, and automation theory.

HOL provides a wide range of proof commands, including rewriting tools and decision procedures. The system is user-programmable which allows proof tools to be developed for specific applications; without compromising reliability [36].

The set of types, type operator, constants, and axioms available in HOL are organized in the form of theories. There are many theories, which are arranged in a hierarchy, have been added to axiomatize lists, products, sums, numbers, primitive recursion, and arithmetic. On top of these, users are allowed to introduce application-dependent theories by adding relevant types, constants, axioms, and definitions.

The HOL system supports higher order logic with three main expressions:

- Variables can range over functions and predicates.
- The logic is typed.
- There is no separate syntactic category of formulae.

The HOL syntax contains syntactic categories of types and terms whose elements are intended to denote respectively certain sets and elements of sets. The types of the HOL logic are expressions that denote sets (in the universe $\mathcal{U}$). There are four kind of types in HOL logic. Type variables stand for arbitrary sets in the universe, they are part of the meta-language and are used to range over object language types. Atomic types denote fixed sets in the universe. For example, the standard atomic types *bool* denotes the distinguished two-element set 2. Compound types have the form $(\sigma_1, \cdots, \sigma_n)op$, where $\sigma_1, \cdots, \sigma_n$ are the argument types and op is a type operator of arity n. Type operators denote operations for constructing sets. Function types denote the set of all (total) functions from the set denoted by its domain to the set denoted by its range [32].

The terms of the HOL logic are expressions that denote elements of the sets denoted by types. There are four kinds of terms in HOL logic. The variables are sequences of letters or digits beginning with a letter. The constants have the same syntax as variables, but stand for fixed values. The function applications or combinations have the general form $t_1(t_2)$ where t_1 is called the operator and t_2 is the operand. The result of such a function application can itself be a function. The lambda terms (λ-terms) or abstractions denote for functions. Such a term has the form $\lambda x.t$ (where t is a term) and denotes the function f defined by $f(x) = t$. The syntax and semantics of the particular logical system supported by HOL notation used in this paper is summarized in Table 2.1. Note that the cons infix operator (::) is used to represent an enumerated list ($hd :: tl$) and the ($\hat{}t$) notation is used to instantiate the value of the term t as shown in the bottom of the table.

Kind of term	*HOL notation*	*Standard notation*	*Description*
Truth	T	$\top$	true
Falsity	F	$\bot$	false
Negation	t	$\neg t$	not t
Disjunction	$t_1 \backslash / t_2$	$t_1 \vee t_2$	t_1 or t_2
Conjunction	$t_1 / \backslash t_2$	$t_1 \wedge t_2$	t_1 and t_2
Implication	$t_1 ==> t_2$	$t_1 \Longrightarrow t_2$	t_1 implies t_2
Equality	$t_1 = t_2$	$t_1 and t_2$	t_1 equal t_2
$\forall$-quantification	$!x.t$	$\forall x.t$	for all x:t
$\exists$-quantification	$?x.t$	$\exists x.t$	for some x:t
ϵ-term	$@x.t$	$\epsilon x.t$	an x such that:t
Conditional	if t then t_1 else t_2	$(t \rightarrow t_1, t_2)$	if t then t_1 else t_2
List Type	$h :: t$	$[h; t]$	[hd;tl]
Antiquotations	$\hat{}t$	$\hat{}t$	Evaluates to the ML value of t

Table 2.1: Terms of the HOL Logic

The basic interface to the system is a Standard Meta Language (SML) interpreter. SML [65] is both the implementation language of the system and the Meta Language in which proofs are written. The HOL system supports two main different proof methods: forward and backward proofs in a natural-deduction style calculus.

In forward proof, the steps of a proof are implemented by applying inference rules chosen by the user, and HOL checks that the steps are safe. All derived inference rules are built on top of a small number of primitive inference rules. This approach has some limitations since it is hard to know where to state the proof and, for large proofs, to determine which sequence of rules to apply. The results are strong and the user can have great confidence since the most primitive rules are used to prove a theorem.

In backward proof, the user sets the desired theorem as a goal. Small programs written in SML called tactics and tacticals are applied to break the goal into a list of subgoals. Tactics and tacticals are repeatedly applied to the subgoals until they can be resolved. In practice, forward proof is often used within backward proof to convert each goal's assumptions into a suitable form.

Theorems in the HOL system 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 system 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. In this way, the ML type system protects the HOL logic from arbitrary construction of a theorem, so that every computed value of the type-representing theorem is a theorem. The user can have a great deal of confidence in the results of the system.

The applications of the HOL system can be found in hardware verification, reasoning about security, verification of fault-tolerant computers, and reasoning about real-time systems. It is also used in compiler verification, program refinement calculus, software and algorithms verification, modeling, and automation theory.

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.

2.2 Multiway Decision Graphs

2.2.1 Formal Logic

The formal logic underlying MDG is many-sorted First Order Logic (FOL). The vocabulary consists of *sorts, constants, variables, and function symbols* or (*operators*). Constants and variables have sorts. An n-ary function symbol ($n > 0$) has a type $\alpha_1 \times \alpha_2 \times \cdots \times \alpha_n \rightarrow \alpha_{n+1}$, where $\alpha_1 \cdots \alpha_{n+1}$ are sorts. 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. A signal of an abstract sort represents a data signal.

The enumeration of a concrete sort α is a set of distinct constants of sort α. We refer to constants occurring in enumerations as *individual constants*, and to other constants

as *generic constants*. An individual constant can appear in the enumeration of more than one sort α, and is said to be of sort α for each of them. Variables and generic constants, on the other hand, have unique sorts.

The *terms* and their *types* (*sorts*) are defined inductively as follows: a constant or a variable of sort α; and if f is a function symbol of type $\alpha_1 \times \alpha_2 \times \cdots \times \alpha_n \rightarrow \alpha_{n+1}$, where $\alpha_1 \cdots \alpha_{n+1}$, $n \geq 1$, and $A_1, \cdots, A_n$ are terms of types $\alpha_1 \cdots \alpha_{n+1}$, then $f(A_1 \cdots A_{n+1})$ is a term of type α_{n+1}. A term consisting of a single occurrence of an individual constant has multiple types (the sorts of the constant) but every other term has a unique type.

We say that a term, variable or constant is concrete (resp. abstract) to indicate that it is of concrete (resp. abstract) sort. A term is *concretely reduced* iff it contains: (i) the individual constants; (ii) the abstract generic constants; (iii) the abstract variable; and (iv) the terms of the form $f(A_1 \cdots A_{n+1})$ where f is an abstract symbol and $A_1, \cdots, A_n$ are concretely-reduced terms. Thus, the concretely-reduced terms are those that have no concrete sub terms other than individual constants. A term of the form $f(A_1 \cdots A_{n+1})$ where f is a cross-operator and $A_1, \cdots, A_n$ are concretely-reduced terms is called *cross-term*. An *equation* is an expression $A_1, \cdots, A_n$ where A_1 and A_n are terms of same type α. *Atomic formulae* are the equations, plus **T** (truth), and **F** (falsity). Formulae are built from the atomic formulae in the usual way using logical connectives and quantifiers.

An *interpretation* is a mapping Ψ that assigns a denotation to each sort, constant and function symbol such that:

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

2. If α is a concrete sort with enumeration $a_1, a_2, \cdots, a_n$ then
   $\Psi(\alpha) = \Psi(a_1), \Psi(a_2), \cdots, \Psi(a_n)$ and $\Psi(a_i) \neq \Psi(a_j)$ for $1 \leq i \prec j \leq n$.

3. If c is a generic constant of sort α, then $\Psi(c) \in \Psi(\alpha)$. If f is a function symbol of type $\alpha_1 \times \alpha_2 \times \cdots \times \alpha_n \rightarrow \alpha_{n+1}$ then $\Psi(f)$ is a function from cartesian product $\Psi(\alpha_1) \times \cdots \Psi(\alpha_n)$ into the set $\Psi(\alpha_{n+1})$.

Let X be a set of variables, a *variable assignment* with domain X compatible with an interpretation Ψ is a function φ that maps every variable $x \in X$ of sort α to an element $\varphi(x)$ of $\Psi(\alpha)$. We write Φ_X^Ψ for the set Ψ-compatible assignments to the variables in X, $\Psi, \varphi \models P$ if P denotes truth under an interpretation Ψ and a Ψ-compatible variable assignment φ to the variables that occur free in P, and $\models P$ if a formula P denotes truth under every interpretation Ψ and every Ψ-compatible variable assignment to the variables that occur free in P. Two formulae P and Q are *logically equivalent* iff $\models P \Leftrightarrow Q$.

2.2.2 Abstract State Machines

Abstract description of State Machines (ASMs) is a model used for describing hardware designs at the Register Transfer Level (RTL). It was introduced by Corella et al. [21, 91]. In MDGs, a state machine is described using finite sets of input, state and output variables, which are pair-wise disjoint. The behavior of a state machine is defined by its transition/output relations including a set of reset states. Using ASMs, a data value can be represented by a single variable of abstract type, rather than by a vector of Boolean variables, and a datapath operation is represented by an uninterpreted function symbol. As ROBDDs are used to represent sets of states and transition/output relations for finite state machines (FSM), MDGs are used to compactly encode sets of (abstract) states and transition/output relations for ASMs. This technique replaces the implicit enumeration technique [22] with the implicit abstract enumeration [21, 91].

2.2.3 Structure

MDGs are graph representation of a class of quantifier-free and negation-free first-order many sorted formulae. It subsumes the class of Bryant's (ROBDDs) [13] 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)[21], such that:

1. Every leaf node is labeled by the formula T, except if the graph G has a single node, which may be labeled T or F.

2. 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.

Then, a graph G can be viewed as representing a formula defined inductively as follows: (i) if G consists of a single leaf node labeled by a formula P , then G represents P; (ii) if G has a root node labeled A with edges labeled $B_1, \cdots, B_n$ leading to subgraphs $G'_1, \cdots, G'_n$, and if each G'_i represents a formula P'_i, then G represents the formulae $\bigvee_{1 \leq i \leq n}((A = B_i) \wedge P_i)$.

Figure 2.1 shows two MDGs example 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 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).

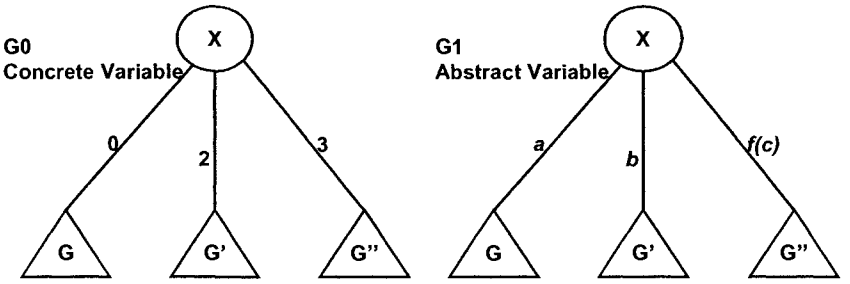


Figure 2.1: Example of Multiway Decision Graphs Structure

MDGs represent and manipulate a certain subset of first order formulae, which we call Directed Formulae (DFs). DFs can represent the transition and output relations of a state machine, as well as the set of possible initial states and the sets of states that arise during reachability analysis.

The MDG operations and verification procedures are packaged as a tool and implemented in Prolog [20]. We show below the basic MDG operations:

Conjunction Operation: The conjunction operation performs conjunction for two DFs not having any abstract variables in common.

Relational Product Operation (RelP): The RelP operation performs conjunction and existential quantifying for a two DFs. It is used for image computation.

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

Pruning By Subsumption Operation (PbyS): The PbyS operation used to approximate the logical difference operation between two sets represented as DF. It removes all the paths of a DF P from another DF Q.

2.2.4 The MDG-Tool

The MDG-tool [90] provides facilities for invariant checking, verification of combinational circuits, sequential verification, equivalence checking of two state machines and model checking.

The input language of the MDGs tool is a Prolog-style hardware description language called (*MDG-HDL*) [21], which supports structural specification, behavioral specification or a mixture of both. A structural specification is usually a netlist of components connected by signals, and a behavioral specification is given by a tabular representation of transition/output relations or a truth table.

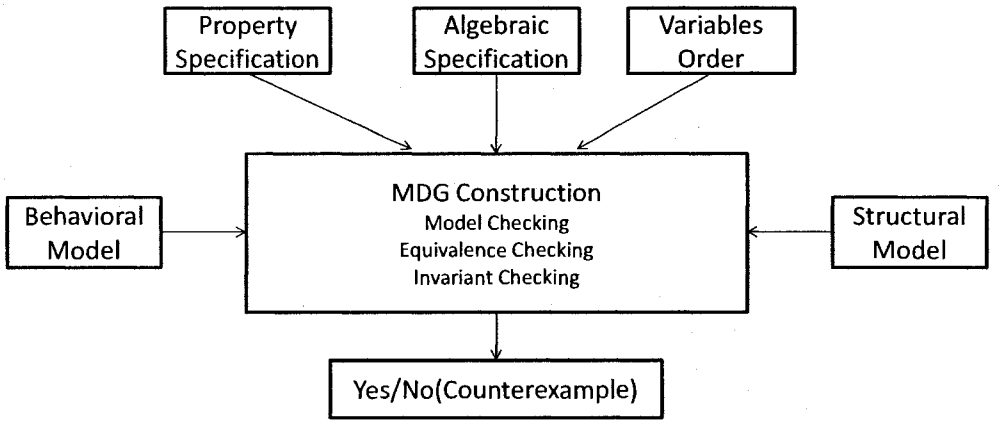


Figure 2.2: The Structure of the MDGs-tool

As shown in Figure 2.2, in order to verify a design with the tool, we first need to specify it in MDG-HDL (design specification and design implementation). Moreover, an algebraic specification is to be given to declare sorts, function types, and generic constants that are used in the MDG-HDL description. Rewrite rules that are needed to interpret function symbols should be provided here as well. Like for ROBDDs, a symbol order according to which the MDG is built could be provided by the user. This symbol order can affect critically the size of the generated MDG. Otherwise,

MDG can use an automatic dynamic ordering.

2.2.5 MDGs Model Checking

The MDGs model checking is based on an abstract implicit state enumeration. The circuit to be verified is expressed as an Abstract State Machine (ASM) and properties to be verified are expressed by formulae in $\mathcal{L}_{MDG}$ [88]. The ASM describes digital systems under verification at a higher level of abstraction.

$\mathcal{L}_{MDG}$ atomic formulae are Boolean constants (True and False), or equations of the form $(t_1 = t_2)$, where t_1 is an ASM variable (input, output or state variable) and t_2 is either an ASM system variable, an individual constant, an ordinary variable or a function of ordinary variables. Ordinary variables are defined to memorize the values of the system variables in the current state. The basic formulas (called *Next_let_formulas*) in which only the temporal operator **X** (next time) is defined as follows [6]:

- Each atomic formula is a *Next_let_formulas*;
- If p, q are *Next_let_formulas*, then so are: !p (not p), p&q (p and q), p|q (p or q), p $\rightarrow$ q (p implies q), **X**p (next-time p) and LET (v=t) IN p, where t is a system variable and v an ordinary variable.

Using the temporal operators **AG** (*always*), **AF** (*eventually*) and **AU** (*until*), the supported $\mathcal{L}_{MDG}$ properties are defined in the following BNF grammar:

$$
\begin{aligned}
\text{Property} ::= \; & A(Next\_let\_formula) \\
& | \, AG(Next\_let\_formula) \\
& | \, AF(Next\_let\_formula) \\
& | \, A(Next\_let\_formula) \, U \, (Next\_let\_formula) \\
& | \, AG(Next\_let\_formula) \Rightarrow F(Next\_let\_formula) \\
& | \, AG((Next\_let\_formula) \Rightarrow \\
& \qquad ((Next\_let\_formula) \, U \, Next\_let\_formula)))
\end{aligned}
$$

Model checking in the MDGs system is carried out by building automatically additional circuit that represents the $Next\_let\_formulas$ appearing in the property to be verified, compose it with the original circuit, and then check a simpler property on the composite machine [88].

Chapter 3

Formalization of MDG Syntax

In this chapter, we describe the way we used to represent the transition relation from graph representation to Directed Formulae DF. Then, we justify the embedding of the DF and the well-formedness conditions in HOL. Finally, we provide an example to illustrate our embedding.

3.1 Transition Relation: Graph or Formula

Different approaches have been used to formalize transition relations either as terms and formulae or as Directed Acyclic Graphs (DAGs). The first is a formal logic representation using data type definitions [7, 35], while the latter is a graphical representation using trees and graphs [64, 85].

First of all, the graph is represented as a data structure in the theorem prover. This representation should reflect the abstract properties of graphs and should be flexible to be suitable for different domains and for many applications to model complex designs. Several examples can be cited: to model communication networks (railway

track network [8]), also in transport industry, the problem of finding the most economical route of delivering goods and the problem of maximizing the network capacity can be solved using graphs.

Chou [15] gradually formalized a considerable amount of graph theory in the HOL theorem proving. The theory of undirected graphs is formalized in HOL notions as the empty graphs, single-node graphs, finite graphs, subgraphs, paths, reachability, acyclicity, trees, subtrees, and merging disjoint subgraph of a graph. Based on this formalization, the correctness of distributed algorithms is verified in HOL [16].

Ridge [72] mechanized some results concerning graphs and trees. His formalization is very close to that found in [15]. The edges are sets of vertices in the case of Ridge while [15] takes edges as atomic objects, and uses an incidence relation to describe when an edge connects two vertices. The main objective of the work is to be able to handle infinite graphs and trees.

Modeling the decision diagram as a decision tree or graph is motivated by reducing memory space and computation time needed to build a BDD: by eliminating redundancy from the canonical representations as described by [64, 85]. The main difficulties are caused by data structure sharing and by the side-effects resulted in the computation. The algorithms usually mark the processed nodes or store the results calculated for a subtree or subgraph in a hash-table to avoid recalculation. The definition of such a mechanism is quite complex for automatic reasoning. The advantage of course is that there is a little work in this area so probably much scope for research.

On the other hand, modeling the transition relations as terms and formulae is smoother for proofs especially those based on induction. Also, in applications like model checking, one would deal with several terms, and any efficient implementation must define sharing. The work presented in [7, 35, 37, 44] is an example of the logical

approach.

The choice between the two approaches depends on the objectives. If we want to reason about the implementation itself and its correctness, then its better to define transition relations 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 of the algorithms, 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.

3.2 Embedding Directed Formulae in HOL

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

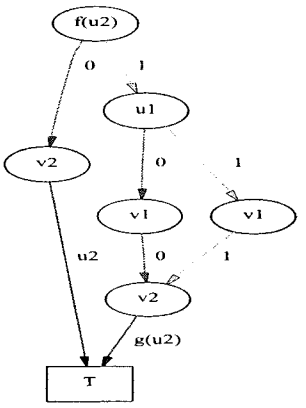
Sort $\mathcal{S}$	::=	$S \mid \underline{S}$
Abstract Sort S	::=	$\alpha \mid \beta \mid \gamma \mid \cdots$
Concrete Sort $\underline{S}$	::=	$\underline{\alpha} \mid \underline{\beta} \mid \underline{\gamma} \mid \cdots$
Generic Constant C	::=	$a \mid b \mid c \mid \cdots$
Concrete Constant $\underline{C}$	::=	$\underline{a} \mid \underline{b} \mid \underline{c} \mid \cdots$
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.

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).

In order to illustrate the MDG, we consider the following example DF of type $\{u_1, u_2\} \rightarrow$

$\{v_1, v_2\}$, where u_1 and v_1 are variables of a concrete sort *bool* with enumeration $\{0, 1\}$ while u_2 and v_2 are variables of an abstract sort α, g is an abstract function symbol of type $\alpha \rightarrow \alpha$ and f is a cross-operator of type $\alpha \rightarrow bool$. Then, the Figure below shows the MDG representing this example as well as its corresponding DF formula.



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

Using HOL recursive datatype, the MDG sorts are embedded using two constructors called `Abst_Sort` and `Conc_Sort`. This is declared in HOL as follows:

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

The `Abst_Sort` takes as argument an abstract sort name of type *alpha* (which means that the sort is actually abstract and hence can represent any HOL type). For example, if *wordn* is an abstract sort, then it is defined in HOL as:

$\vdash_{def}$ `wordn = Abst_Sort "wordn"`

The `Conc_Sort` takes a concrete sort name and its enumeration of type *string* as an input argument. For example, if *bool* is a concrete sort with ["0";"1"] as enumeration, then it is defined in HOL as:

$\vdash_{def}$ `bool = Conc_Sort "bool" ["0";"1"]`

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 are either of concrete or abstract sort. An individual constant can have multiple sorts depending on the enumeration of the sort, while an 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 as follows:

```
Ind_Cons ::= Ind_Cons of string → 'alpha Sort
Gen_Cons ::= Gen_Cons of string → 'alpha Sort
```

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. As shown below, *oone* is defined as an individual constant, *max* is defined as a generic constant and m is defined as an abstract variable:

```
⊢def  oone = Ind_Cons "1" bool
⊢def  max = Gen_Cons "max" wordn
⊢def  m = Abst_Var "m" wordn
```

Similarly, we use some predicates to determine whether a constant is an individual or a generic and a variable is concrete or abstract.

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. Abstract function is declared using `Abst_Fun` constructor and the `Cross_Fun` constructor is used to declare a cross function.

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

If *eqz_Fun* is a cross-function takes m as an input and produces a concrete output of sort *bool*, then, *eqz_Fun* is defined as:

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

We have defined a data type D_F . The DF can be True or False or a disjunction of conjunction of equations. Equations are defined as an equality of Left Hand Side (LHS) and Right Hand Side (RHS) based on the DF grammar given earlier and could be one of the following cases:

- LHS is a concrete variable = RHS is an individual constant
- LHS is an abstract variable = RHS is a cross-function, or abstract variable or generic constant.
- LHS is a cross-function = RHS is an individual constant

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_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
```

DF1, DISJ, CONJ1, Eqn, and CONJ are distinct constructors and the constructors EQUAL1, EQUAL2, EQUAL3, EQUAL4, and 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.

Internally, the DF is implemented as a list to simplify the checking of well-formedness conditions and the embedding of MDG operations. However, this representation is completely transparent for the user of the embedded MDG operations later. Then it is sufficient

to input the DF as formulae and the transformations (proved correct) is done automatically. The DF representation as a list having the following format:

$$[\overbrace{[\overbrace{[lhs_{11}; rhs_{11}]}^{eq_{11}}; \cdots ; \overbrace{[lhs_{1n}; rhs_{1n}]}^{eq_{1n}}]}^{disjunct_1}; \cdots ; \overbrace{[\overbrace{[lhs_{m1}; rhs_{m1}]}^{eq_{m1}}; \cdots ; \overbrace{[lhs_{mn}; rhs_{mn}]}^{eq_{mn}}]}^{disjunct_m}]$$

where a DF is given as:

$$\begin{aligned} \text{DF} \quad = \quad & eq_{11} \wedge eq_{12} \wedge \cdots \wedge eq_{1n} \bigvee \\ & eq_{21} \wedge eq_{22} \wedge \cdots \wedge eq_{2n} \bigvee \\ & \vdots \\ & eq_{m1} \wedge eq_{m2} \wedge \cdots \wedge eq_{mn} \end{aligned}$$

We extract the DF using the `STRIP_DF_list` function:

$\vdash_{def}$ `(STRIP_DF_list (DF1 (CONJ1 (CONJ E M))) =`

`[(both_side_eq E)] :: STRIP_DF_list(DF1 (CONJ1 M))) ∧`

`(STRIP_DF_list (DF1 (DISJ (Eqn E) D)) =`

`[(both_side_eq E)] :: STRIP_DF_list(DF1 D) ) ∧`

`(STRIP_DF_list (TRUE) = [[["TRUE"]]] ) ∧`

`(STRIP_DF_list (FALSE) = [[["FALSE"]]]) ∧`

`(STRIP_DF_list (DF1 a) = STRIP_DISJ_list a)`

`STRIP_DISJ_list` function is used to extract each disjunct and store it in a list, while `STRIP_CONJ_list` function is used to extract both side of equations and store them in the inner sublist. Similarly, `STRIP_Fun` function is used to extract the arguments of abstract and cross functions and store them in a list. The HOL definitions of the mapping functions and the well-formedness conditions are included in Appendix A. This mapping simplifies our implementation and enables us to automate MDG operations by using the infrastructure of the predefined *List Theory* in HOL to inherit all definitions and theorems.

On the other hand, we defined a `STRIP_INV_DF` function (Appendix A) to map lists to DF format. We proved a theorem to show that our mapping from any well-formed DF to list format and from lists to DF is correct as shown by the following theorem:

Theorem 3.2.1 *DF Mapping Correctness*

DF Mapping Correctness $\vdash$ $\forall$ *df. Is_Well_Formed_DF df* $\Longrightarrow$

STRIP_INV_DF (STRIP_DF_list df) = df)

PROOF:

The proof is conducted by structural induction on df. □

3.3 Well-formedness Conditions

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 [21] 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 is no need of 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.

The predicate `Is_Well_Formed_DF` is defined as:

$\vdash_{def}$ `∀df. Is_Well_Formed_DF df =`

```
        Condition2 (STRIP_DF df) ∧
        Condition3 (FLAT(STRIP_ABS_DF df)) (STRIP_DF df))
```

where `Condition2` and `Condition3` represent the well-formedness conditions (Appendix A.1). `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 derive a set of inference rules which guarantees if a given DF is well formed according to the definition given before. The recursive data type package automatically returns the following theorems which characterize each condition separately. Table 3.1, gives these inference rules since it is more adequate to DF, and independent from the logic of HOL. We translate these inference rules as theorems in HOL.

Table 3.1: Well-Formedness (WF) Inference Rules

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

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

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

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

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

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

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

The well-formedness conditions can be summarized as:

- Condition 1: The condition is satisfied by construction following the DF syntax. The axiom `WF_E1` represents the equality between a concrete variable and a concrete individual 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 theorems needed for the inference rule representing this condition are given by:

```
WF_True: ⊢ Is_Well_Formed_DF (TRUE)
WF_False: ⊢  Is_Well_Formed_DF (FALSE)
WF_E1: ⊢  ∀Conc_Var Ind_Con.
   Is_Well_Formed_DF (DF1(CONJ1(Eqn(EQUAL1 Conc_Var Ind_Con))))
WF_E2: ⊢  ∀Abst_Var Abst_Fun.
   Is_Well_Formed_DF (DF1(CONJ1(Eqn(EQUAL2 Abst_Var Abst_Fun))))
WF_E3: ⊢  ∀Crs_Fun Ind_Con.
   Is_Well_Formed_DF
        (DF1(CONJ1(Eqn(EQUAL3 Crs_Fun Abs_Var Ind_Con))))
WF_E4: ⊢  ∀Abst_Var Abst_Var.
   Is_Well_Formed_DF (DF1(CONJ1(Eqn(EQUAL4 Abst_Var Abst_Var))))
WF_E5: ⊢  ∀Abst_Var Gen_Con.
   Is_Well_Formed_DF (DF1(CONJ1(Eqn(EQUAL5 Abst_Var Gen_Con))))
```

- Conditions 2: We add the LHS (`left_eq` function) which extracts the Left Hand Side variable of a given equation. The assumptions needed for this condition are: the two well-formed equations and the LHS of each equation should not be equal. We prove a HOL theorem (`WF_Conj`) that states the correctness of the inference rule related to this condition:

```
WF_Conj: ⊢  ∀E1 E2. (Is_Well_Formed_DF(DF1(CONJ1(Eqn E1))) ∧
    Is_Well_Formed_DF(DF1(CONJ1(Eqn E2)))) ∧ ¬(left_eq E1=left_eq E2) ⟹

        Is_Well_Formed_DF (DF1(CONJ1(CONJ E1 (Eqn E2))))
```

- Condition 3: The assumptions needed for this condition are: the two well-formed conjuncts and the abstract variables of each conjunct should be equal. We prove the theorem (`WF_Disj`) that states the correctness of the inference rule related to this last condition:

```
WF_Disj: ⊢  ∀conj1 conj2. Is_Well_Formed_DF (DF1(CONJ1 conj1)) ∧
  Is_Well_Formed_DF (DF1(CONJ1 conj2)) ∧
  (FLAT(STRIP_ABS_DF) (DF1(CONJ1 conj1))=FLAT(STRIP_ABS_DF (DF1(CONJ1 conj2))))
  ⟹  Is_Well_Formed_DF (DF1(DISJ conj1 (CONJ1 conj2)))
```

We have implemented a HOL tactic to automate the checking of well-formedness conditions based on the function `Is_Well_Formed_DF` defined above.

3.4 MIN-MAX Example

We consider the MIN-MAX circuit described in [21]. The MIN-MAX state machine shown in Figure 3.1 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.

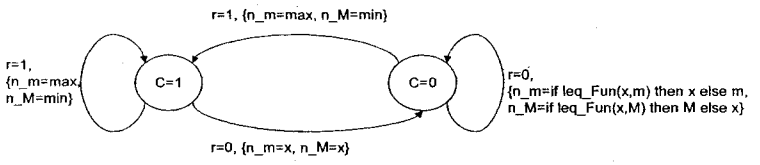


Figure 3.1: MIN-MAX State Machine

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 DFs of these individual transition relations, for a particular custom symbol order, are shown below:

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

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 and cross-function.

```
⊢def  bool = Conc_Sort "bool" ["0";"1"]
⊢def  wordn = Abst_Sort "wordn"
⊢def  zzero = Ind_Cons "0" 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
```

Also, We define some equations and disjuncts:

```
⊢def  eq1 = EQUAL1 ^r ^zzero
⊢def  eq4 = EQUAL1 ^n_c ^oone
⊢def  eq9 = EQUAL3 ^leq_Fun [^x;^m] ^zzero
⊢def  eq11 = EQUAL5 ^n_m ^max
⊢def  mdg1  = CONJ ^eq2 (CONJ ^eq4 (CONJ ^eq11 (Eqn ^eq16)))
⊢def  mdg2 = CONJ ^eq1 (CONJ ^eq5 (CONJ ^eq3  (CONJ ^eq7
                                        (CONJ ^eq13 (Eqn ^eq9)))))
```

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                                          (3.1)
```

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

An example of applying (`WF`) inference rules given in Table 3.1, is presented below. 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.

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

where:

$eq2 = (r = oone)$

$Cond2 = (LHS(eq2) \neq LHS(eq4) \neq LHS(eq11) \neq LHS(eq16))$

$Cond3 = (Abst\_Var(eq2 \wedge eq4 \wedge eq11 \wedge eq16) = Abst\_Var(mdg2 \vee mdg3 \vee mdg4 \vee mdg5 \vee mdg6))$

Chapter 4

Formalization of 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, the outputs will not necessarily be a well-formed DF. Also, as the DF represent a canonical graph, the variables order must be preserved, which is not satisfied when applying HOL operations. Our embedding is built to address specifically these concerns. In this chapter, we provide a formal definitions of MDG basic operations, the correctness and the well-formedness proof. The detailed embedding can be found in [4].

4.1 The Conjunction Operation

The conjunction operation is performed over MDG structures; examples of MDG conjunction is shown in Figure 4.1. In F1, the two top variables of P and Q are the same concrete variables or cross-terms. In F2, the top variable of P is a concrete variable or cross-term A, and order(A) < order(top variable of Q). Finally in F3, P and Q have different primary abstract variables, and order(A) < order(top variable

of Q). However, it is not a well formed MDG (canonical), and T_i must be substituted for A having secondary occurrences in Q.

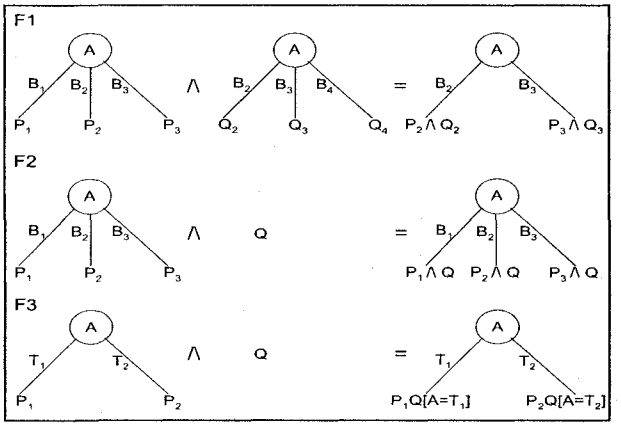


Figure 4.1: The conjunction operation

In terms of DF's, 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 $R = \textbf{Conj}\ (\{P_i\}_{1 \leq i \leq 2})$ 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:

$$\models R \Leftrightarrow (\bigwedge_{1 \leq i \leq 2} P_i) \tag{4.1}$$

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.

4.1.1 The Conjunction Constraints:

There are two sets of constraints over the conjunction operation: one set is needed to be respected in order to execute the algorithm (pre-conditions), and the second is related to the execution nature of the algorithm itself. The first takes as inputs a set of well-formed DFs and a list that represents the union of the DF order lists. The second type can be summarized as:

1. If the two DFs have a common root label of abstract sort, the conjunction of the two DFs cannot be computed. For example, it is easy to see that there is no DF representing a formula that is logically equivalent to $(x = a) \wedge (x = b)$, if x is an abstract variable and a and b are distinct generic constants. This case never occurs in the reachability analysis algorithm, because the conjunction is always performed between DFs having mutually disjoint sets of primary abstract variables.

2. If the root label of df1 comes before the root label of df2, and the label of df1 is an abstract variable x, the resulting DF may not be well-formed. This is justified because x may have secondary occurrences (LHS) in df2, and hence it may have secondary occurrences in the result. Therefore, x may have both primary (RHS) and secondary occurrences in the result, contradicting the well-formedness conditions. Symmetrically, the same situation for df2.

The result of the operation must be a well-formed DF representing the conjunction of df1 and df2 (post-condition). Thus, it suffices to eliminate x from df2 by substitution for x in df2 or replacing the secondary occurrences of x in df2 with the respective terms.

The outlined method for computing the conjunction 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.

4.1.2 The Conjunction Embedding:

In the next step, we define the conjunction operation in HOL. The operation 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 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 using the DF_CONJ function. Note that "_" means "don't care".

```
⊢def  (DF_CONJ [] _ _ L3 = [] ) ∧
      (DF_CONJ _ [] _ L3 = [] ) ∧
      (DF_CONJ (hd1::tl1) (hd2::tl2) L1 L3 =
         DF_CONJUNCTION (hd1) (hd2::tl2) L1 L3 :: DF_CONJ (tl1) (hd2::tl2) L1 L3 )
```

The DF_CONJUNCTION function determines the conjunction of the first disjunct of df1(disj1_df1) and df2 as shown in line 5 and defined as:

```
⊢def  (DF_CONJUNCTION [] _ _ L3 = [] ) ∧
      (DF_CONJUNCTION _ [] _ L3 = [] ) ∧
      (DF_CONJUNCTION (hd1::tl1) (hd2::tl2) L1 L3 =
         if (IS_EL [] (HD_DISJUNCT (hd1::tl1) (hd2) L3) )  then
             DF_CONJUNCTION (hd1::tl1) (tl2) L1 L3
         else
             HD_SUBST (HD_DISJUNCT (hd1::tl1) (hd2) L3) L1 ::
                 DF_CONJUNCTION (hd1::tl1) (tl2) L1 L3 )
```

The HD_DISJUNCT function determines the conjunction between the first disjunct of both DFs (lines 6 to 8). Then, we apply the substitution by taking the disjunct and check that 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. For instance, in the case of abstract edge label $(x = y) \wedge (y = a)$ the resulting substitution is $(x = a)$. The substitution is carried out using the HD_SUBST function:

```
⊢def  (HD_SUBST [] = [] ) ∧
      (HD_SUBST L = SPLIT1 L L )
⊢def  (SPLIT1 _ [] (hd3::tl3) = [] ) ∧
      (SPLIT1 [] _ (hd3::tl3) = [] ) ∧
      (SPLIT1 (hd1::tl1) (hd2::tl2) []=SPLIT (hd1::tl1) (hd2::tl2)) ∧
      (SPLIT1 (hd1::tl1) (hd2::tl2) (hd3::tl3) =
         if (IS_EL (HD(TL hd1)) (HD_list (hd2::tl2))) then
             if ( HD(TL hd1) = (HD hd2) ) then
                 (HD hd1::(TL hd2))::SPLIT1 tl1 (hd2::tl2) (hd3::tl3)
             else
                 SPLIT1 (hd1::tl1) (tl2) (hd3::tl3)
         else if (IS_EL(HD( hd1)) (FLAT(FLAT(TL_list(hd3::tl3))))) then
             SPLIT1 tl1 (hd2::tl2) (hd3::tl3)
         else
             hd1 :: SPLIT1 tl1 (hd2::tl2) (hd3::tl3) )
```

Line 9 recursively appends the result and moves 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. The detailed algorithm describing the HD_DISJUNCT function is given in Algorithm 2.

The HD_DISJUNCT function tests if the two equations of the two disjuncts have the same order, by checking the position of the head of both equations (lines 1 and 2) using position function. Line 3 adds the equation to the result and move to the next equation, in both disjuncts, and call HD_DISJUNCT recursively (line 4). Otherwise if the head of both equations are equal but the tail (RHS) are not equal, then the result will be empty and we stop and move to the next disjunct in df2 (lines 5 and 6). If the first equation of df1 comes before df2, then append it to the result and move to the next equation in the same disjunct and repeat the process recursively (lines 8 to 10). Otherwise, if the first equation of df2 comes before df1, then append the equation of

Algorithm 2 HD_DISJUNCT (disj1_df1, disj1_df2, L)

```
1: if (position(LHS(Eq1_df1),L) = position(LHS(Eq1_df2),L)) then
2:   if (RHS of both Eqs are equal) then
3:     append Eq1 to the result;
4:     call HD_DISJUNCT(tail(disj1_df1), tail(disj1_df2), L);
5:   else
6:     empty the list and quit the HD_DISJUNCT;
7:   end if
8: else if (position(LHS(Eq1_df1),L) < position(LHS(Eq1_df2),L)) then
9:   append Eq1_df1 to the result;
10:  call HD_DISJUNCT(tail(disj1_df1), disj1_df2, L);
11: else
12:  append Eq1_df2 to the result;
13:  call HD_DISJUNCT(disj1_df1, tail(disj1_df2), L);
14: end if
```

df2 to the result and repeat the process recursively (lines 11 to 13). The `HD_DISJUNCT` function is defined in HOL:

```
⊢def  (HD_DISJUNCT [] L2 L3 = L2) ∧
      (HD_DISJUNCT (hd1::tl1) [] L3 = (hd1::tl1)) ∧
      (HD_DISJUNCT (hd1::tl1) (hd2::tl2) L3 =
        if (HD hd1 = HD hd2) then
            if ( TL hd1 = TL hd2) then
                hd1 :: HD_DISJUNCT  tl1 tl2 L3
            else
                [] :: HD_DISJUNCT  [] [] L3
        else if (position L3 (HD hd1)<position L3 (HD hd2)) then
            hd1 :: HD_DISJUNCT tl1 (hd2::tl2) L3
        else
            hd2 :: HD_DISJUNCT (hd1::tl1)  tl2 L3 )
```

where the `position` function is used to check the order as described in Appendix A.2.

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 mentioned that the function STRIP_DF_list is used to translate a DF into a list format, while the function STRIP_Fun is used to extract the arguments of cross-term and store them in a list.

Example

The following example is used for illustration. The circuit consists of two abstract variables *count* and *n_count*, three symbol functions *inc_Fun*, *dec_Fun* and *eqz_Fun*. The *inc_Fun* and *dec_Fun* take as an input *count* of abstract sort and produce *inc_Fun(count)* and *dec_Fun(count)* an abstract output. The cross-function *eqz_Fun* takes *count* as an input and produces a concrete output of sort *bool*. The input y of the circuit and *count* is the output of the circuit as represented by MDG1. MDG2 represent another circuit similar to MDG1 in addition to an input f of concrete sort and an abstract variable w as shown in Figure 4.2. The order list given as: [”f”;”y”;”*eqz_Fun*”;”w”;”*n_count*”].

Our objective is to apply the conjunction on df1 and df2. We defined the directed formulae df1 and df2 representing MDG1 and MDG2 in HOL, respectively.

```
val df1 = ``DF1(DISJ ^mdg1 (DISJ ^mdg3 (CONJ1 ^mdg8)))``
val df2 = ``DF1 (DISJ ^mdg7 (DISJ ^mdg9 (CONJ1 ^mdg10)))``
```

After this step, applying the CONJ_ALG on both DFs will result:

[[[”f”;”0”];[”y”;”0”];[”w”;”*inc_Fun*”];[”*n_count*”;”*inc_Fun*”]];

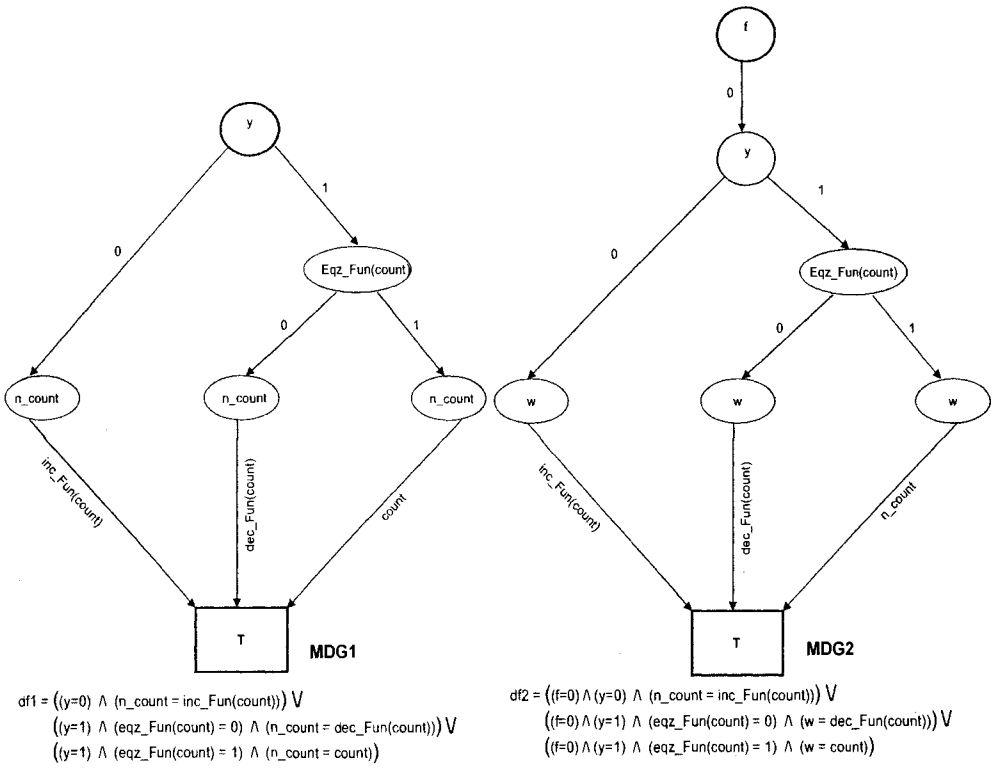


Figure 4.2: MDG1 and MDG2

[["f";"0"];["y";"1"];["eqz_Fun";"0"];["w";"dec_Fun"];["n_count";"dec_Fun"]];
[["f";"0"];["y";"1"];["eqz_Fun";"1"];["w";"$count$"];["n_count";"$count$"]]]
and its MDG is shown in Figure 4.3:

4.2 The Relational Product (RelP) Operation

The RelP operation is used to compute the sets of states reachable in one transition from one sets of states (image computation). It combines conjunction and existential quantification.

In terms of DF's, the RelP takes as inputs two DFs P_i, $1 \leq i \leq 2$, of types $U_i \rightarrow V_i$ and a set of variables E to be existentially quantified, and produces a DF

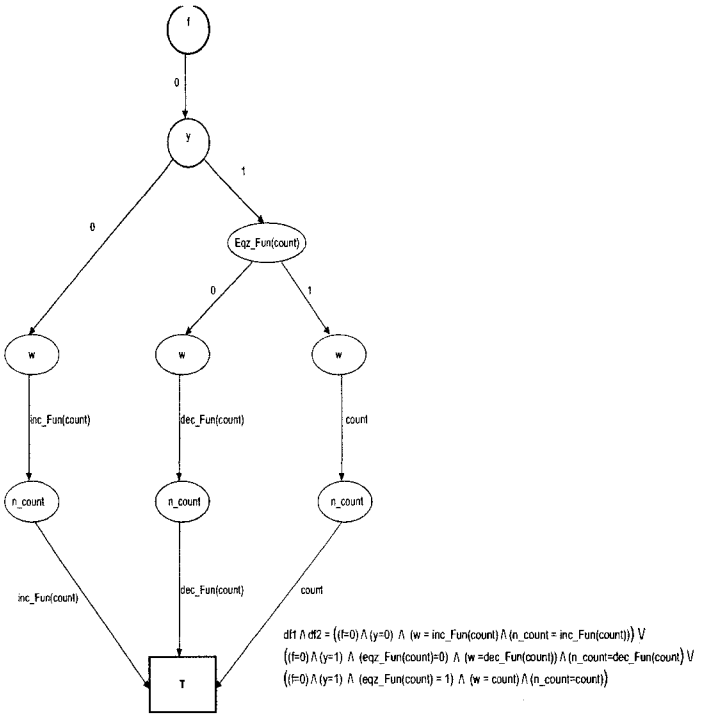


Figure 4.3: MDG1 CONJ MDG2

$R =$ **RelP** $(\{P_i\}_{1 \leq i \leq 2}, E)$ such that:

$$\models R \Leftrightarrow ((\exists E)(\bigwedge_{1 \leq i \leq 2} P_i)) \tag{4.2}$$

The operation computes the conjunction of the P_i and existentially quantifies the variables in E. For $1 \leq i < j \leq 2$, V_i and V_j must not have any abstract variables in common. The result of computing conjunction and existentially quantification would be a DF of type $(\bigcup_{1 \leq i \leq 2} U_i) \backslash (\bigcup_{1 \leq i \leq 2} V_i) \rightarrow ((\bigcup_{1 \leq i \leq 2} V_i) \backslash E)$.

4.2.1 The RelP Constraints:

The RelP constraints will be the same as mentioned in the conjunction operation. A new condition is added: the set of variables to be existentially quantified must

be primary variables of at least one of the DFs. The result of the operation must be a well-formed DF representing the conjunction of df1 and df2 and existentially quantifies with respect to the set of variables (post-condition).

4.2.2 The RelP Embedding:

In order to formalize the RelP operation, we are going to use the embedded conjunction operation in Section 4.1 to get the conjunction of two DFs. Then we embed the extra condition regarding the set of variables E to be existentially quantified over the result of the conjunction operation by calling the function `EXIST_QUANT` as shown below:

```
⊢def  (EXIST_QUANT [] (hd2::tl2) = []) ∧
      (EXIST_QUANT (hd1::tl1) [] = (hd1::tl1) ) ∧
      (EXIST_QUANT (hd1::tl1) (hd2::tl2) =
          EXIST_QUANT (EXIST_QUANT1 (hd1::tl1) [ hd2]) tl2 )
```

where `EXIST_QUANT1` is a function used to quantify one variable over the conjunction result. Then, the `RelP` function is defined as:

```
⊢def  ∀df1 df2 L1 L2. (RelP_ALG df1 df2 L1 L2 =
                  EXIST_QUANT (CONJ_ALG df1 df2 L1) L2)
```

Using the embedding of the conjunction operation simplifies the formalization and shows the reusability of our embedding and proof.

Example

Lets back to the example in Section 4.1.2. The set of variables to be quantified is given as: ["*f*";"*n_count*"]. Our objective is to apply the relational product on df1 and df2. Applying the RelP_ALG on both df1 and df2 will result:

[["y";"0"];["w";"inc_Fun"]];

[["y";"1"];["eqz_Fun";"0"];["w";"dec_Fun"]];

[["y";"1"];["eqz_Fun";"1"];["w";"$count$"]]]

and its MDG is shown in Figure 4.4:

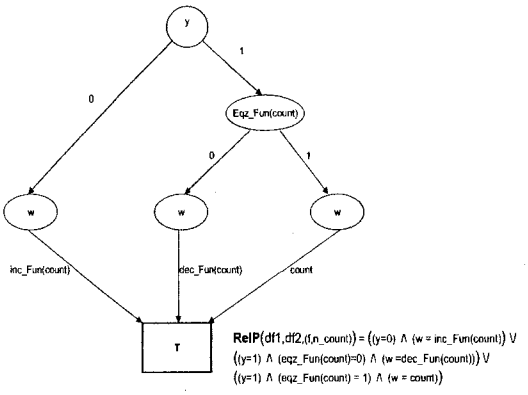


Figure 4.4: MDG1 RelP MDG2

4.3 The Disjunction Operation

The example shown in Figure 4.5 is for two MDGs with the same root label. We add a redundant concrete edge which requires that the same abstract primary variable is present along all paths. In terms of DF's, the disjunction operation takes as inputs

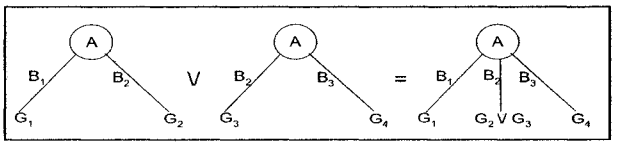


Figure 4.5: The disjunction operation

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:

$$\models R \Leftrightarrow (\bigvee_{1\leq i\leq 2} P_i) \tag{4.3}$$

The operation computes the disjunction of its n inputs in one pass and note that this 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 $\models R \Leftrightarrow (P_1 \vee P_2)$.

4.3.1 The Disjunction Constraints:

Again, we have two types of constraints. The first type is the same as described in the conjunction constraints. The second type can be summarized in the following items:

1. The abstract variables in both DFs must be the same otherwise, the disjunction cannot be computed. This condition can be checked by comparing the abstract variables in any disjunct of both DFs, since both DFs are well-formed. For example, there is no DF representing a formula that is logically equivalent to $(x = a) \vee (y = b)$, where x and y are abstract variables and a and b are distinct generic constants.

2. If the two DFs have different root labels, but the label that comes first in the node-label order is a concrete variable x or a cross-term, we can revert to the case where the labels are the same by adding a redundant node labeled x at the top of the DF (not labeled by x).

Indeed, the roots of DFs must have the same label, otherwise the label that comes first in node-label ordering must be a concrete variable or cross-term; then the disjunction operation is applied recursively. The result of the algorithm must be a well-formed DF representing df1 and df2 disjunction (post-condition).

4.3.2 The Disjunction Embedding:

Similarly, we embed the disjunction operation in HOL as we did in the conjunction operation. The detailed algorithm is given in Algorithm 3.

Algorithm 3 DISJ_ALG (df1, df2, L)

```
1: if terminal DF then
2:    return result;
3: else if (STRIP_ABS1_DF df1 = STRIP_ABS1_DF df2) then
4:    DF_DISJUNCTION(df1, df2, L)
5: else
6:    return empty list;
7: end if
```

Algorithm 3 starts with two well-formed DFs and an order list L. The resulting 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 the abstract variables in both DFs. If they are equal, then (line 4) determines the disjunction of two DFs by calling `DF_DISJUNCTION`. Otherwise, the algorithm returns empty list (line 6). The function `DF_DISJUNCTION` is defined in HOL as given below:

```
⊢def  (DF_DISJUNCTION (hd1::tl1) [] L = [] ) ∧
      (DF_DISJUNCTION [] (hd2::tl2) L = [] ) ∧
      (DF_DISJUNCTION (hd1::tl1) (hd2::tl2) L =
            union (FLAT(DF_DISJUNCT1 (hd1::tl1) (hd2::tl2) L))
                  (FLAT(DF_DISJUNCT1 (hd2::tl2) (hd1::tl1) L)))
```

where the function `DF_DISJUNCT1` applies the conjunction to find any similarity between the two DFs. Otherwise, it adds the disjuncts of df1 that does not appear in df2. `DF_DISJUNCT1` is similar to `HD_DISJUNCT` used in conjunction operation. Again we re-apply `DF_DISJUNCT1` on df2 and df1 to cover all disjuncts of both DFs. Then, we take the union of `DF_DISJUNCT1`(df1, df2, L) and `DF_DISJUNCT1`(df2, df1, L), where `DF_DISJUNCT1` is defined as:

```
⊢def  (DF_DISJUNCT1 [] (hd2::tl2) L = []) ∧
      (DF_DISJUNCT1 (hd1::tl1) [] L = [] ) ∧
      (DF_DISJUNCT1 (hd1::tl1) (hd2::tl2) L =
         if ((position L (HD (HD hd1))) = (position L (HD (HD hd2)))) then
               DF_DISJUNCT (hd1::tl1) (hd2::tl2) L
         else if ((position L (HD(HD hd1))) < (position L (HD (HD hd2)))) then
               (DF_DISJUNCT (hd1::tl1) (FLAT(APPEND_LIST
                (UNION_HD_list(HD_list(hd1::tl1))) (hd2::tl2))) L)
         else
             (DF_DISJUNCT (FLAT(APPEND_LIST
                 (UNION_HD_list(HD_list(hd2::tl2))) (hd1::tl1))) (hd2::tl2) L))
```

The disjunction operation is defined in HOL as:

```
⊢def  ∀df1 df2 L. DISJ_ALG df1 df2 L =
  (if (df1 = TRUE) ∨ (df2 = TRUE) then [[["TRUE"]]]
  else (if (df1 = FALSE) ∧ (df2 = FALSE) then [[["FALSE"]]]
  else (if df1 = FALSE then STRIP_DF_list df2
  else (if df2 = FALSE then STRIP_DF_list df1
  else (if FLAT (STRIP_ABS_DF df1) = FLAT (STRIP_ABS_DF df2) then
           UNION_HD_list (DF_DISJUNCTION(STRIP_DF_list df1) (STRIP_DF_list df2)L)
  else [])))))
```

Example

The following example is a subset of the circuit description in Section 4.1.2. Given MDG1 and MDG2 as shown in Figure 4.6.

The order list given as: ["f";"y";"eqz_Fun";"n_count"]. Our objective is to apply the disjunction on both DFs. Applying the DISJ_ALG on both DFs will result: [[["f";"0"];["y";"0"];["n_count";"inc_Fun"]];

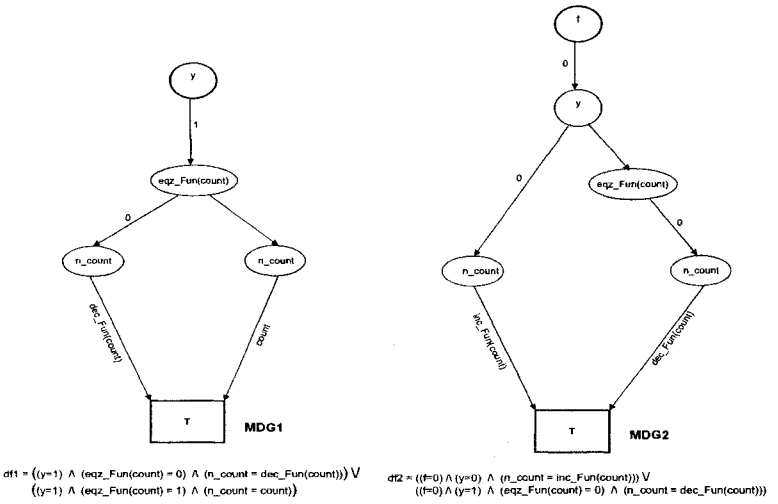


Figure 4.6: MDG1 and MDG2

[["f";"0"];["y";"1"];["eqz_Fun";"0"];["n_count";"dec_Fun"]];

[["f";"0"];["y";"1"];["eqz_Fun";"1"];["n_count";"$count$"]]]

and its MDG is shown in Figure 4.7.

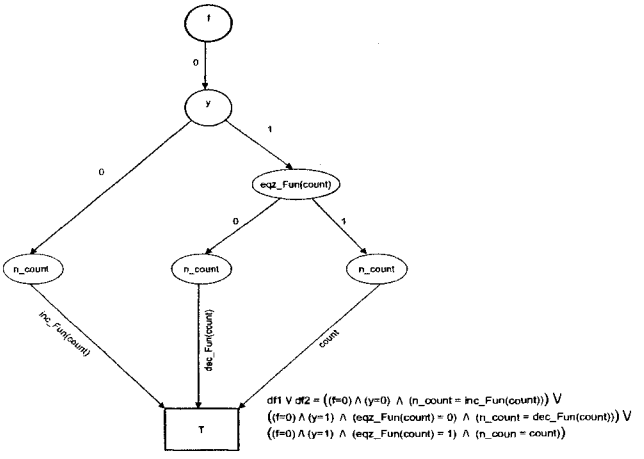


Figure 4.7: MDG1 DISJ MDG2

4.4 The Pruning by Subsumption (PbyS) Operation

The pruning by subsumption operation is used in checking set inclusion (fixed point detection and in invariant checking); *Frontier* set simplification. In Figure 4.8 an example is shown: $Q = PbyS(NS, R)$. In NS, the path of the concrete variable c, labeled by 0, is subsumed by R. Thus, it will be removed from Q. The other path, labeled by 1, cannot be removed and appears in Q. Informally, it removes all the paths of N from Q. In terms of DF's, the PbyS takes as inputs two DFs P and Q

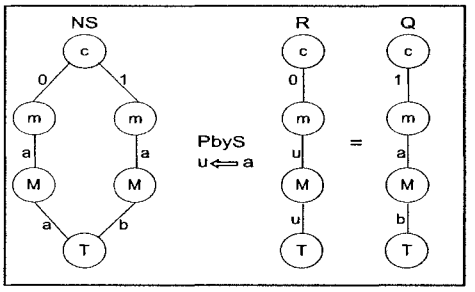


Figure 4.8: The PbyS operation

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 $R = \mathbf{PbyS}\ (P, Q)$ of type $U \rightarrow V_1$ derivable from P by *pruning* (i.e. by removing some of the disjoints) such that:

$$\models R \vee (\exists E)Q \Leftrightarrow P \vee (\exists E)Q \tag{4.4}$$

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 (4.4), 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 $(\exists E)Q$, this approximation may lead to non-termination problem (see [88] for more details). In general, there is no DF logically equivalent to $P \wedge \neg(\exists E)Q$. If R is F, then it follows tautologically from (4.4) that $\models P \Rightarrow (\exists E)Q$.

4.4.1 The PbyS Constraints:

Unlike the previous operations, 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).

4.4.2 The PbyS Embedding:

The embedding of the PbyS is explained in Algorithm 4. It starts with two well formed DFs and order list L. The resulting 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

Algorithm 4 PbyS_ALG (df1, df2, L)

```
1: if 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
```

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). The DF_PbyS function is defined recursively in HOL as given below:

```
⊢def  (DF_PbyS [] _ _ _ _ L = [] ) ∧
      (DF_PbyS (hd1::tl1) [] _ _ _ L =  (hd1::tl1)) ∧
      (DF_PbyS (hd1::tl1) (hd2::tl2) _ (hd4::tl4) (hd5::tl5) L =
      (DF_PbyS (hd1::tl1) (hd2::tl2) _ (hd4::tl4) (hd5::tl5) L =
            if ((FLAT(hd4::tl4)=[])=(FLAT(hd5::tl5)=[])) then
                DF_PbyS (hd1::tl1) (hd2::tl2) [] [] [] L
            else
                PbyS_1 (hd1::tl1) (hd2::tl2) (hd4::tl4) (hd5::tl5)) ∧
      (DF_PbyS (hd1::tl1) (hd2::tl2) _ [] [] L =
            if (IS_EL hd1 (hd2::tl2))  then
                DF_PbyS tl1 (hd2::tl2) [] [] [] L
            else
                hd1 :: DF_PbyS tl1 (hd2::tl2) [] [] [] L)
```

The DF_PbyS function has two main cases:

- The top symbol of df1 is not included in the symbols of df2, then df2 will not

subsumed df1.

- The top symbol of df1 and df2 are the same or the top symbol of df1 is included in the symbols of df2. We have three cases:
  - The common top symbol is a concrete variable, then its individual constant (RHS) of every equation of df1 must be the same or included in df2, otherwise it will not subsumed by df2.
  - The common top symbol is an abstract variable, then its (RHS) will be either abstract variable, generic constant or abstract function. In this case, df1 will be subsumed by df2 with suitable substitution for the RHS and the arguments of the abstract function as specified in `PbyS_1` function. It checks the existence of the first disjunct of df1 in df2. If it exists then the function will discard it (subsumed by df2). Otherwise the disjunct is added to the result (cannot be subsumed):

```
⊢def  (PbyS_1 [] (hd2::tl2) _ _ = [] ) ∧
      (PbyS_1 (hd1::tl1) [] _ _ =  (hd1::tl1)) ∧
      (PbyS_1 (hd1::tl1) (hd2::tl2) L4 (hd5::tl5) =
        if (PbyS_2 hd1 (hd2::tl2) L4 (hd5::tl5) = [])  then
           PbyS_1 tl1 (hd2::tl2) L4 (hd5::tl5)
        else
           PbyS_2 hd1 (hd2::tl2) L4 (hd5::tl5) ::
           PbyS_1 tl1 (hd2::tl2) L4 (hd5::tl5))
```

    The function `PbyS_2` is defined in Appendix A.5.
  - The common top symbol is cross-operator, then its individual constant

(RHS) of every equation of df1 must be the same or included in df2, otherwise it will not subsumed by df2. Note that, the arguments of the cross-operator might be substituted.

Finally, the pruning by subsumption operation is:

```
⊢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
            []
```

4.4.3 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 Table 4.1, in each run we increase the size of the disjunct and measure the execution time.

Table 4.1: The PbyS Performance

Disjunct No.	Execution Time (sec)
1	6.3
2	6.67
3	7
4	7.3
8	8.3
16	9.3
32	11.3
64	13
128	14.3
182	15.3

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 disjuncts.

Figure 4.9 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.

4.5 The Correctness Proof

In general, we have two kinds of theorems: one theorem regarding the correctness proof of each MDG operation, and the other one for preserving the well-formedness of the operation results. The correctness represents a mathematical proof of consistency

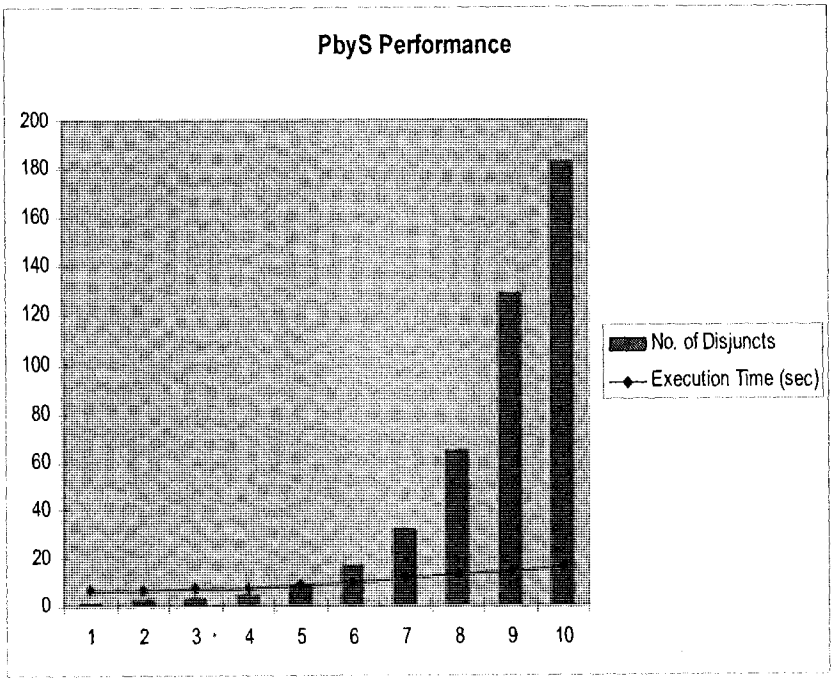


Figure 4.9: The PbyS Performance

between the operation specification and its implementation in HOL, while the well-formedness to ensure the obtained MDG representation is canonical.

Theorem 4.5.1 *Operation Correctness*

ASSUME:

1. df1 and df2 are well-formed DF.

2. L is an order list equal to the union of df1 and df2 order lists.

Then, the MDG operation of df1 and df2, and HOL logical operation of df1 and df2, are equivalent.

Operation Correctness $\vdash \quad \forall$ *df1 df2.* $\exists L.$ *Is_Well_Formed_DF df1* $\wedge$

Is_Well_Formed_DF df2 $\wedge$ *(ORDER_LIST df1 df2* $=$ *L)* $\Longrightarrow$

(Logical_HOL_Opr df1 df2 $=$ *MDG_Opr_HOL df1 df2 L)*

PROOF:

By structural induction on df1 and df2 and rewriting rules. The goal is to prove the equivalence of MDG operation and HOL logical operation for these DF. However, the proof strategy is systematic as shown in Figure 4.10. It consists of feeding the same inputs to the logical HOL predefined operations and to the embedded MDG operations. The output of the embedded MDG operation `MDG_Opr_HOL` is well-formed DF. This DF will be compared with the output formulae of logical HOL operation `Logical_HOL_Opr`. We check the equivalence of both and prove it as a theorem using structural induction and rewriting rules. Moreover, this goal generates hundreds of subgoals since the proof takes all the cases of DF. The terminal cases are directly proved by applying the rewriting rule. Many base cases are generated, for example, in the case of the conjunction operation, the proof when both df1 and df2 are just an equation is shown by Lemma 1 in Appendix A.2. □

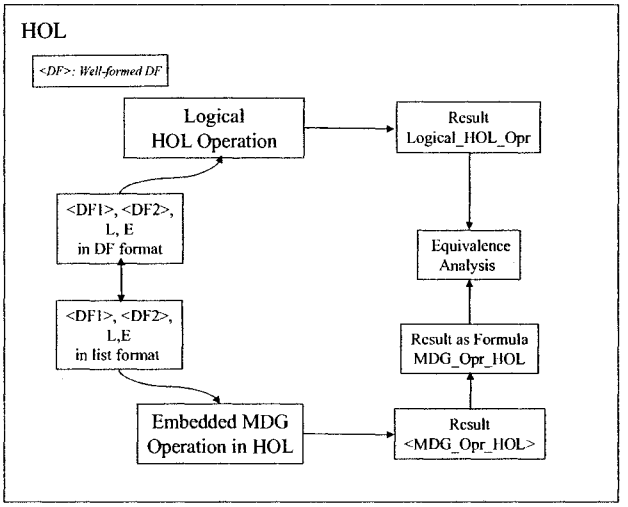


Figure 4.10: Correctness Methodology

Lets back to our conjunction example in Section 4.1.2. It is easy to prove the correctness of the result by instantiating Theorem 4.5.1 to prove the below goal:

```
∀ df1 df2. ∃L. ((L= "f";"y";"eqz_Fun";"n_count";"w"]) ∧
        (df1=^df1) ∧ (df2=^df2)) ⟹
        (Is_Well_Formed_DF df1) ∧ (Is_Well_Formed_DF df2) ∧
        (ORDER_LIST df1 df2 = L) ⟹
        ((CONJ_LOGIC df1 df2 L) = DISJ_LIST(CONJ_ALG df1 df2 L) )
```

Theorem 4.5.2 *Well-formedness Preservation*

ASSUME:

1. df1 and df2 are well-formed DF.

2. L is an order list equal to the union of df1 and df2 order lists.

Then, the result of the MDG operation of df1 and df2 is well-formed.

Preserving Well-formedness ⊢ ∀ *df1 df2.* ∃*L. Is_Well_Formed_DF df1* ∧ *Is_Well_Formed_DF df2* ∧ *(ORDER_LIST df1 df2 = L)* ⟹ *(Is_Well_Formed (MDG_Opr_HOL df1 df2 L))*

PROOF:

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

4.6 Embedding and Proof Discussion

Technical difficulties may raise at this stage. To respect the formal logic of HOL, the formalization of the Directed Formulae in [88] has been modified. The new DF formalization is more suitable for HOL and avoid potential infinite loops. It ensures the reachability analysis termination when it should occur [6]. In fact, applying

induction on *DF*, with this modifications, ameliorate the reasoning with the MDG structure in HOL. This is one of the contributions of our work. Also, because the number of subgoals generated is big, modifying one definition may change the flow of the proof as shown in Appendix A.2. Finally, the proof goes through all the operational definitions of each operation and gives us more confidence about our embedding.

In fact, the conjunction operation has consumed most of the proof preparation effort. Most of the definitions and proofs were reused in the proof of 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.

Moreover, some useful properties can be proved based on our formalization. For example, to check whether a set of states is a subset of another set of states, we used the PbyS operation to prove this equality as shown in Lemma 2 in Appendix A.5. Other properties can be proved over the conjunction and disjunction operations such as associativity and commutativity.

Chapter 5

Formalization of MDG Reachability Analysis

In this chapter, 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 review the MDG reachability analysis [21]; followed by its definition in HOL along with a discussion on the technical challenges.Then, we use the MIN-MAX as an illustrative example for our reachability analysis embedding. Finally, a set of benchmarks has been conducted to ensure the applicability and the performance of the MDG-HOL platform.

5.1 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 call *Abstract State Machine* a state machine given an abstract description in terms of DFs, or equivalently MDGs, as defined in [21].

Definition 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 description of the output relation.

Algorithm 5 shows how the analysis of the reachable states of M can be performed based on the abstract description D.

Algorithm 5 MDG Reachability Analysis

1: $R := IS$;
2: $Q := IS$;
3: $i := 0$;
4: **while** $Q \neq F$ **do**
5: $i := i + 1$;
6: $IN := new\_inputs(i)$; - `Produce new inputs`
7: $NS := next\_states(IN, Q, Tr)$; - `Compute next state`
8: $Q := frontier(NS, R)$; - `Set difference`
9: $R := union(R, Q)$; - `-- 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.

5.2 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 circuits. However, we defined a conversion which returns a goal to be proven interactively using induction for a given circuit (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 (Appendix A.6) by calling the recursive function `RA_n` with the circuit 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 circuit. The result is obtained using the DF relational product operator `RelP(Q,Tr)`.

- 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. 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.

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

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) Circuit_Parameters = RA_n (n) Circuit_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 circuit and cannot be considered as a general solution because of the non-termination problem. Indeed, the abstract representation and the uninterpreted function symbol may not lead the reachability analysis algorithm to terminate [21]. Thus, the MDG reachability computation is theoretically unbounded. Meanwhile, several practical solutions have been proposed to solve the non-termination problem. The authors in [6] related the problem to the nature of the analyzed circuit. Furthermore, they have characterized some mathematical criteria that leads explicitly to non-termination of particular classes of circuits.

The reachability analysis conversion is general and can be applied to any DF of a circuit. What will change are only the DF and the set of initial states, if we consider the order list is given. The conversion shown by Algorithm 6 encapsulates the following steps:

Algorithm 6 Re_An Conversion (I, Q, Tr, E, Ren, L, R)

1: Formalize the circuit 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 circuit 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 only for 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 circuits. Furthermore, the selection of `n0` is based on the knowledge and the heuristic of the circuit since the

induction is not explicitly identified as illustrated by the MIN-MAX example.

5.3 Example: The MIN-MAX revisited

The MIN-MAX state machine has two input variables: $I = [[[\text{"}x\text{"};\text{"}r\text{"}]]]$, set of initial states:

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

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

Then, we applied the reachability analysis conversion steps mentioned in Algorithm 6:

The first step: formalize the MIN-MAX circuit in terms of DF and check the well-formedness conditions by applying the predicate `Is_Well_Formed_DF` (conversion tactic) on (Tr, Q, R) as shown in 3.4.

The second step: we apply only one `Reach_Step` to compute the next reachable state as explained in Algorithm 6, 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)
```

The base step is straightforward and the induction step is carried out by rewriting rules.

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

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

5.4 The MDG-HOL Platform

We support our platform by experimental results executed on different benchmarks. Indeed, our results shows that such an embedding offers a considerable gain compared to the automatic approach of model checking tool. However, a huge investment in time should be spent in developing the theory and proving the necessary theorems in theorem provers.

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) [63], and one large benchmark: Island Tunnel Controller (ITC) [91]. The performance is measured in terms of full reachability analysis for these models. Tables 5.1

and 5.2 compare the number of nodes, number of functions the memory usage, reachability analysis time, and human effort generated by MDG-HOL and FormalCheck (V2.3) [17] model checking, respectively, run on a Sun enterprize server with Solaris 5.7 OS and 6.0 GB memory. The reachability time is measured in FormalCheck by estimating the average reachability time for the set of all properties associated with the design.

Table 5.1: 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 5.2: 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

Discussion

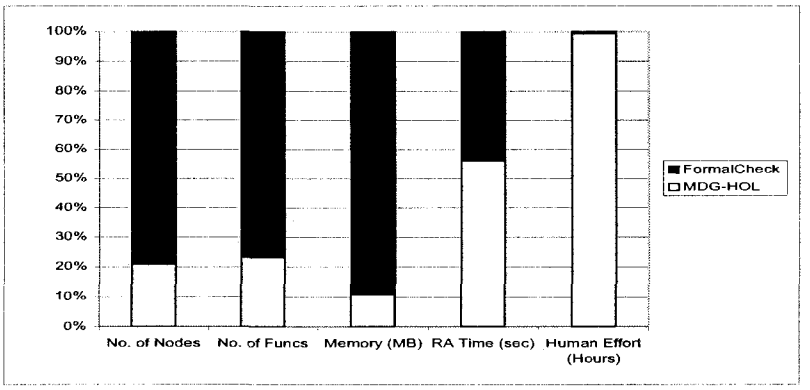


Figure 5.1: MDG-HOL and FormalCheck Small Benchmarks

Figure 5.1 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.

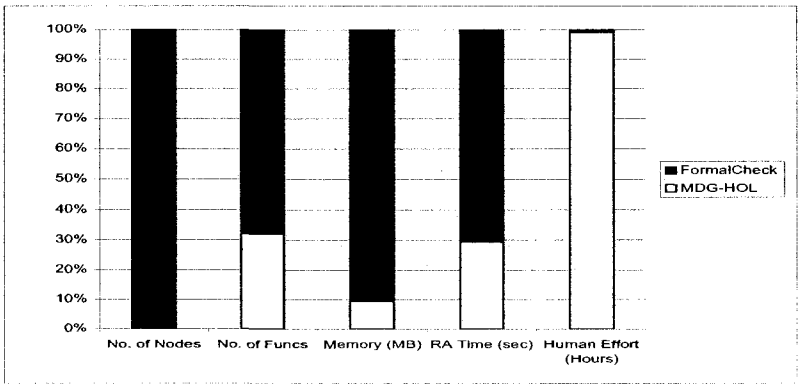


Figure 5.2: MDG-HOL and FormalCheck Big Benchmarks

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 Figure 5.2. 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 human effort since it took almost five days to prove fixpoint for small benchmarks and three weeks for the ITC benchmark compared to few hours using 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.

Chapter 6

Applications and Case Studies

The last decades has seen a remarkable advancement in model checking technology. Still in todays hundreds million gates designs, the size of the over all model is beyond the capability of any model checking tool. The solution is to develop synergies between various verification methodologies, and between design and verification, in order to achieve high level coverage.

Model reduction approaches are used to reduce the model size prior to verification. These approaches are based on abstract interpretation which supports the reduction of a system under verification to a more abstract and smaller one. This means if the property holds for the reduced system, it holds for the original one as well.

The tendency in design methodologies is to define new paradigms based on higher level of abstraction. The design is described using different level of details: system level, process level, communication level before the implementation at RTL level [18]. The assume-guarantee paradigm, among other techniques, is a known reduction technique [19] that has been used in several paradigm to reduce model checking CPU and memory usage.

Paradoxically, the effort and advancement in formal verification tools is, for the first time, against the concept of rising the abstraction level. In fact, most of the efforts today are spent on developing Satisfiability Checking (SAT) based tools to perform several forms of model checking as they are less sensitive to the problem sizes and the state explosion problem of classical BDDs based model checkers. However, the concept of verification by SAT is a pure low level paradigm. In fact, the transition relation is encoded with the property in Conjunctive Normal Form (CNF); every individual bit of very data signal must be encoded by a separate Boolean variable, causing the size of CNF to grow considerably with the number of variables. The next step is to apply a search algorithm that tries to prove the SAT or the UNSAT of the CNF formula through variant and mutation of the Davis-Putnam-Loveland-Logemann (DPLL) algorithm [26]. The limit of SAT solvers is annually pushed with more and more smart tricks and heuristics with SAT solvers competition that feed this tendency. The result is a remarkable advancement that have been achieved by bit-blasting of high level designs into CNF in order to apply SAT as shown in [12].

The center of interest of all these methods is performance, they don't provide any guarantee that the reduction or optimization heuristics are sound and applied correctly. In fact, most of the abstraction techniques steps for SAT bit-blasting or heuristics steps to optimize SAT solvers are not derived though proof theory or logical operations which is considered as a gap between the sound model checking concept and these optimized tools. At this stage, Automatic Theorem Prover (ATP) are more advantageous. In fact, in the ATP each lemma should be derived from existing lemmas and theorems before using it inside the ATP system. However, the use of ATP is still considered difficult and time consuming because of the undecidability problem. Some solutions aims to define model checking automatic verification inside ATP for example

Amjad in [7].

In this chapter, we propose a reduction methodology based on our MDG-HOL platform that combines an automatic high level model checking tool within the HOL theorem prover. The idea is to use the consistency of the specifications to verify if the reduced model is faithful to the original one. We use the MDG-HOL platform to prune the transition relation of the circuits to produce a smaller one that is fed to the MDG model checker. Then, using High Order Logic we check automatically if the reduction technique is soundly applied. The methodology verifies the soundness of the verification output and not the reduction algorithm itself (non-decidable problem).

In Section 6.1, we give a background of the model reduction techniques and the related work to ours. In Section 6.2, we overview the SAT-MDG reduction methodology of the Island Tunnel Controller. Finally, the verification of assume-guarantee in MDG-HOL of the Look-Aside Interface and the Island Tunnel Controller are presented in Section 6.3.

6.1 Model Reduction Techniques

Model reduction techniques attempt to reduce the size of the model to be checked. There has been extensive research on state space reduction either for both hardware and software systems. For example, we cite reduction compositional reasoning [53], the symbolic representation of states and states transitions [82], state abstraction [61], partial order reduction [83], symmetry reduction [27] or hybrid techniques (combinations of these methods). Those issues and others are surveyed in [19].

Another category of techniques are property-based reduction techniques. Such techniques target the property being checked by using it to simplify the design under verification [43]. SAT techniques are lower-level techniques that seek to improve the

execution of the underlying BDDs engine or SAT solver by exploiting the structure of the model and/or the property [45].

The reduction techniques mentioned above come mostly from the model checking world. From theorem proving world and from the point of view of temporal specifications, there are two types of induction that can be applied. One is induction on time, and the other is induction on the data structures. Safety properties in theorem proving are often proven by induction on time. First, one proves that the property holds in the initial states (the base of the induction), and then, assuming that the property holds in some arbitrary state, one proves that all the states in its transition image also satisfy this property (inductive step). Since the original property is rarely inductive (not strong enough to satisfy the inductive step), it is often necessary to strengthen the invariant before it can be proven. In fact, the theorem prover expert manages this step by re-defining the system in an abstract manner where the model is parameterized and reduced in order to ease the induction proof. This task is usually the hardest and the not-automatic step in theorem proving based verification. Nowadays there are few tools that help compute inductive invariants automatically [10, 73].

We have chosen to concentrate on the hybrid reduction techniques. In this direction, Hazelhurst et al. presented in [39] an approach relying on the use of two industrial tools, one for symbolic trajectory evaluation (STE) [80] and one for symbolic model-checking. STE performs user-supplied initialization sequences and produces a parametric representation of the reached states set. The result must be systematically converted into a characteristic function form, before it can be fed to the model-checker [39].

In [77], the authors proposed a technique to construct a reduced MDGs model for circuits described at system level in VHDL. The simplified model is obtained using

a high level symbolic simulator called *TheoSim* [76], and by running an appropriate symbolic simulation patterns. Later, the authors proposed another technique based on SAT solver. They used a rewriting based SAT solver to produce a smaller model that is fed to the MDGs model checker. The work presented in this chapter provides a verification technique based on MDGs operations and the rewriting engine of the HOL theorem prover to verify the soundness of the reduced model.

All these related work concentrate only on the optimization of the model checker performance. Even if some of these reduction techniques has been proven sound themselves, they do not provide any guarantee that the reduced model of a specific circuit is logically compliant to the original non-reduced one. In another word, they don't provide a way to verify that we have applied the reduction technique correctly. Our approach provides an answer for this particular problem by checking the compliance of the original and reduced model inside the theorem prover. According to our knowledge, this is the first time that the theorem prover is used for this objective.

6.2 SAT-MDG Reduction Verification

6.2.1 Boolean Satisfiability

The Boolean Satisfiability (SAT) problem is a well-known constraint satisfaction problem with many applications in computer-aided design, such as test generation, logic verification and timing analysis. Given a Boolean formula, the objective is to either find an assignment of 0-1 values to the variables so that the formula evaluates to true, or establish that such an assignment does not exist. The Boolean formula is typically expressed in CNF, also called product-of-sums form. Each sum term (clause) in the CNF is a sum of single literals, where a literal is a variable or its negation. An n-clause

is a clause with n literals. For example, $(v_i + v'_j + v_k)$ is a 3-clause. In order for the entire formula to evaluate to 1, each clause must be satisfied, i.e. evaluate to 1.

The complexity of this problem is known to be NP-Complete [31]. In practice, most of the current SAT solvers are based on the Davis-Putnam algorithm [26]. The basic algorithm begins from an empty assignment, and proceeds by assigning a 0 or 1 value to one free variable at a time. After each assignment, the algorithm determines the direct and transitive implications of that assignment on other variables, typically called Boolean Constraint Propagation (BCP). If no contradiction is detected during the implication procedure, the algorithm picks the next free variable, and repeats the procedure. A conflict occurs when implications for setting the same variable to both 1 and 0 are produced. Otherwise, the algorithm attempts a new partial assignment by complementing the most recently assigned variable for which only one value has been tried so far. This step is called backtracking. The algorithm terminates either when all clauses have been satisfied and a solution has been found, or when all possible assignments have been exhausted. The algorithm is complete in that it will find a solution if it exists.

The modern SAT solvers GRASP [52] and rel-sat [9] independently contributed techniques for conflict analysis and conflict-driven learning. The SAT solvers SATO [89] and Chaff [59] improved the basic data structures for performing Boolean constraint propagation and making implications, which constitutes the computational core of Davis-Putnam based SAT solvers. The main idea was to avoid visiting all clauses that a variable appears in by keeping track of two watched literals that are non-false in each clause.

6.2.2 Combining SAT and MDG Methodology

The SAT-MDG reduction technique uses an external rewriting based SAT engine developed within Mathematica to simplify DF by applying functional partitioning and synchronization detection [77]. The method starts with a system level design and a set of properties written in $\mathcal{L}_{MDG}$. As shown in Figure 6.1, the transition relation is translated in terms of DF. Then an abstraction technique is applied to create a CNF formula and a set of associated truth assignments constraints is introduced: B_{DF}. During this step a Boolean variables for every clause in the transition relation with suitable arguments (primary variables (LHS) and uninterpreted function arguments). Also additional constraints are specified between clauses with similar arguments in order to be mutual. From the properties, the set of reduction variables is extracted and fed with the B_{DF} to the rewriting based SAT solver which will decide the truth assignment and the implication of this assignment and produce a reduced transition relation: *Reduced* B_{DF}. Then, the *Reduced* B_{DF} is translated to DF reduced transition relation. The obtained DF with the $\mathcal{L}_{MDG}$ properties will be fed to the MDG Model Checker. The formal verification is performed then on this obtained reduced MDG using the existing MDG package.

6.2.3 Abstracting CNF from DF

Algorithm 7 CREATECNFFORMULA(SYSTEM)

```
1: Formula = CreateLogicFormula(System);
2: BoolFormula = replace each term in Formula with a predicate;
3: Infer constraints between predicates;
4: Transform predicate to Boolean variable;
5: CNFFormula = ConvertToCNF(BoolFormula);
6: Return CNFFormula;
```

Algorithm 7 shows a sketch on how to obtain a transition relation in CNF. It

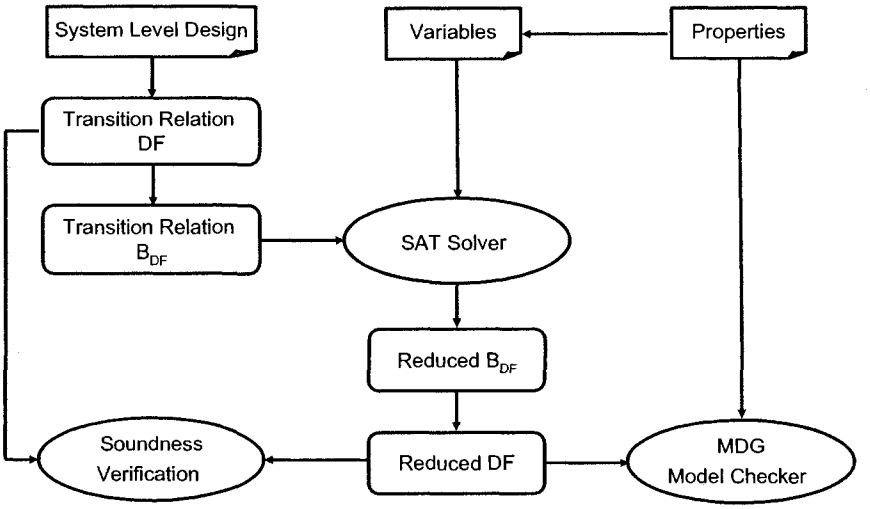


Figure 6.1: Overview of the Methodology

first creates the transition relation in a general format at line 1. Assume the formula is

$$((x = 3) \wedge (y = 2)) \vee ((x = 5) \wedge (y = 4)) \tag{6.1}$$

Line 2 will then introduce n predicates for every clause with LHS argument, so in the above formula four predicates are needed and the formula becomes $(b_1(x) \wedge b_2(y)) \vee (b_3(x) \wedge b_4(y))$. Line 3 introduces additional constraints such that clauses with a similar LHS argument must be mutual. In this example we know that $b_1(x)$ and $b_3(x)$ cannot be true at the same time. Meanwhile, one of them has to be true, otherwise the formula cannot be satisfied $(b_1(x) \oplus b_3(x))$. Similar constraints can be applied to $b_2(y)$ and $b_4(y)$. Therefore, the Boolean formula $B(Tr_{DF})$ and the truth assignment constraints are shown below:

$$\begin{bmatrix} B(Tr_{DF}): & (b_1(x) \wedge b_2(y)) \vee (b_3(x) \wedge b_4(y)) \\ Constraints: & (b_1(x) \oplus b_3(x)) \\ & (b_2(y) \oplus b_4(y)) \end{bmatrix}$$

In line 4, all dependencies are resolved and the predicates will be transformed to Boolean variables (i.e. $b_1(x)$ becomes b_{1x}). Note the Boolean formula is not in CNF yet. There exists linear algorithm to convert any Boolean formula to CNF [84], with additional variables introduced. As mentioned in line 5, the CNF representation for the above formula is:

$$\begin{bmatrix} B(Tr_{DF}): & (b_{1x} \vee b_{3x}) \wedge (b_{2y} \vee b_{3x}) \wedge \\ & (b_{1x} \vee b_{4y}) \wedge (b_{2y} \vee b_{4y}) \\ Constraints: & (b'_{3x} \vee b'_{1x}) \wedge (b_{1x} \vee b_{3x}) \\ & (b'_{4y} \vee b'_{2y}) \wedge (b_{2y} \vee b_{4y}) \end{bmatrix}$$

6.2.4 Extracting Variables from Properties

The approach to select a variable and assign it a value is based on (assumption) extracted from the dependent variables on the property and hence the resulting transition relation will be much smaller. In fact, SAT-MDG approach gives the possibility to assign a concrete variables to the inputs of the system. Thus, an important reduction is gained on the resulting transition relation which improves the performance of the MDG model checker in terms of memory and CPU time.

Just as an example, if we assume that P1 is dependent on b_{1x}, then if the SAT solver decides b_{1x} to be true, then the implication we can get is:

$$\begin{bmatrix} B(Tr_{DF}): & b_{2y} \\ Constraints: & (b'_{4y} \vee b'_{2y}) \wedge (b_{2y} \vee b_{4y}) \end{bmatrix}$$

which represents a very small transition relation consisting of only 1 clause compared to the original one of 4 clauses, and hence improve the performance.

6.2.5 Island Tunnel Controller (ITC)

System Description

The SAT-MDG technique has been demonstrated on the example of the Island Tunnel Controller (ITC) in [91], which was originally introduced by Fisler and Johnson [28].

The *ITC* controls the traffic lights at both ends of a tunnel based on the information collected by sensors installed at both ends of the tunnel: there is one lane tunnel connecting the mainland to an island. At each end of the tunnel, there is a traffic light as depicted in Figure 6.2. There are four sensors for detecting the presence of cars: one at tunnel entrance on the island side (ie), one at tunnel exit on the island side (ix), one at tunnel entrance on the mainland side (me), and one at tunnel exit on the mainland side (mx). In [28], the following constraint is imposed: at most sixteen cars may be on the island at any time. It is assumed that all cars are finite in length, that no car gets stuck in the tunnel, that cars do not exit the tunnel before entering the tunnel, that cars do not leave the tunnel entrance without traveling through the tunnel, and that there is sufficient distance between two cars such that the sensors can distinguish the cars.

As shown in Figure 6.3, the specification of *ITC* is composed of three communication controllers and two counters: The Island Light Controller (ILC), the Tunnel Controller (TC), the Mainland Light Controller (MLC), the Island Counter and the Tunnel Counter (refer to [28] for the state transition diagrams of each component). The *Island Light Controller* (ILC) has four states: *green*, *entering*, *red* and *exiting*. The outputs *igl* and *irl* control the green and red lights on the island side respectively;

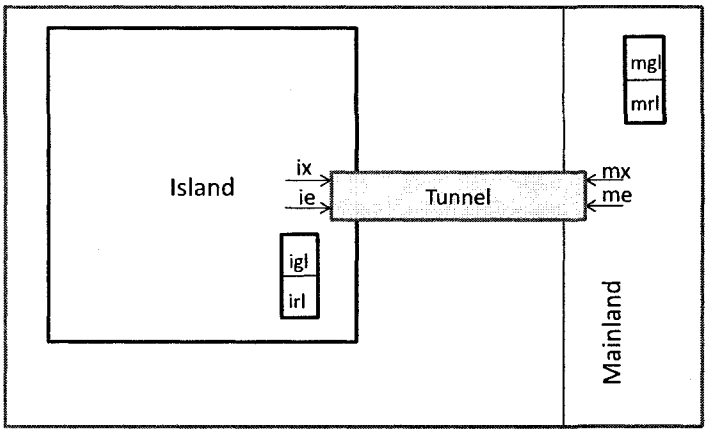


Figure 6.2: The Island Controller

iu indicates that the cars from the island side are currently occupying the tunnel, and *ir* indicates that *ILC* is requesting the tunnel. The input *iy* requests the *ILC* to release control of the tunnel, and *ig* grants control of the tunnel from the island side as shown in Figure 6.3. A similar set of signals is defined for the *Mainland Light Controller* (*MLC*).

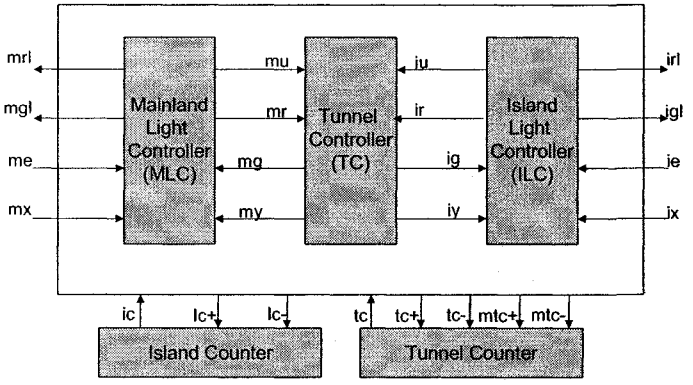


Figure 6.3: Island Tunnel Controller Structure

The *Tunnel Counter* (TC) processes the requests for access issued by *ILC* and

MLC. The *Island Counter* and the *Tunnel Counter* keep track of the car's number currently on the island and in the tunnel, respectively. For the tunnel controller, the counter *tc* is increased by 1 depending on *tc+* or decremented by 1 depending on *tc-* unless it is already 0. The *Island Counter* operates in a similar way, except that increment and decrement depend on *ic+* and *ic-*, respectively: one for the island lights, one for the mainland lights, and one tunnel controller that processes the requests for access issued by the other two controllers.

Verification

Table 6.1 compares the verification results of the original MDG model checking and the reduced one with soundness verification for five properties, run on a Sun enterprize server with Solaris 5.7 OS and 6.0 GB memory. Note that the soundness verification of the PbyS took less than 1 second added to the verification time.

We note that the reduction gain depends on the properties. The best gain in performance is obtained with property P3 where the time is reduced by 6.7 times the original one and the memory is reduced by a factor of 9.3 times. The worst case is the property P1 where the time is reduced by 1.2 times the original one and the memory reduction is not profitable.

In the case of property P1 the assumptions and the functionality tested needs several runs (when using our SAT reduction as case splitting). The sum of these runs for this particular case is a little bit lower to a single run without reduction. For P3, case splitting was really much more efficient. These differences show the sensitivity of the reduction technique to the property verified. Note that the soundness verification using the PbyS operation took less than 10 seconds added to the average time. Despite these fluctuations, the gain average in performance is a factor of 1.4 which is considered

Table 6.1: Comparing the Original MDGs Model Checking Results with the Reduced MC and Soundness Verification Results

Benchmark	*Original MC*			*Reduced MC*		
Properties	**Time**	**Mem**	**Nodes**	**Time**	**Mem**	**Nodes**
P1	65.35	50.1	123080	54.65	47.6	121060
P2	0.12	0.57	263	.10	0.4	211
P3	65.45	48.6	123085	10.73	5.24	12292
P4	65.61	46.4	123082	36.05	26.11	63419
P5	65.89	48.3	123080	49.42	34.95	69966
Average	52.48	38.79	98518	30.19	22.86	53389

as a good result in the case of model checking approaches.

6.3 The Assume-Guarantee Reduction Verification in MDG-HOL

In this section, we present an algorithm to achieve the soundness of the assume-guarantee reduction methodology. First, we present the assume-guarantee reduction methodology and how we generate a mathematical model in terms of DF from the design. Then, we provide the soundness verification of the methodology. Finally, we discuss the MDG-HOL assume guarantee technique for the ITC and LA-1 case studies.

6.3.1 The Assume-Guarantee Reduction Methodology

As shown in Figure 6.4, we generate from the behavioral design written in HDL language an abstract mathematical model in terms of DF.

From a set of properties written in $\mathcal{L}_{MDG}$, we extract the representation of the properties in terms of DF and feed them with the design transition relation to the MDG-HOL platform.

The reduction technique itself could be applied on the HDL description either using HOL or an external tool. However, the result of the reduction should be embedded in HOL as DF or will be translated to MDG-HDL (the input language of MDGs tool).

Then, the verification of the reduction soundness algorithm is applied using MDG operations and the rewriting engine of the HOL theorem prover. If the reduction is proved sound then the formal verification can be performed on the obtained reduced model.

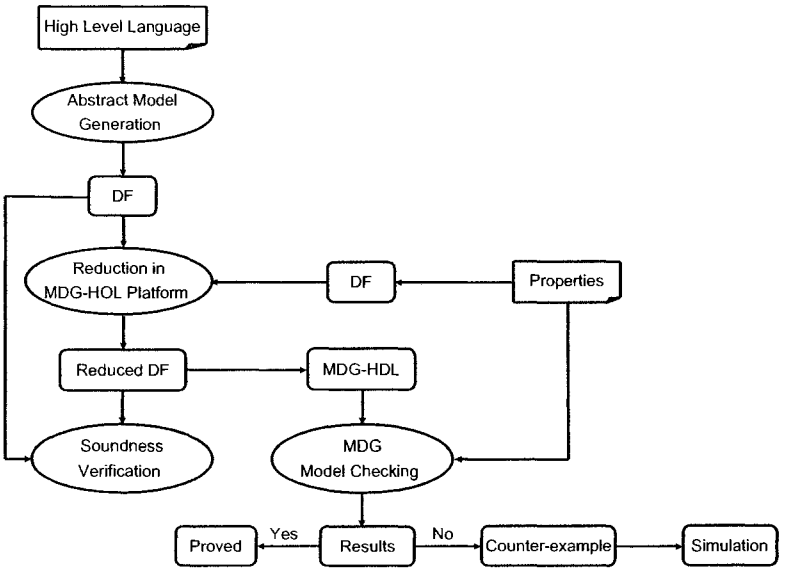


Figure 6.4: Overview of the Reduction Methodology

6.3.2 Generation of Directed Formulae

From High Level Language

When the model is too large, one may use abstraction to reduce the size of the model [24]. Some constraints are removed from the original model, making the model smaller (it has fewer constraints on the transitions). Therefore, if a property is true in the abstract model, then it must be true in the original model as well. Otherwise, a false negative is raised. In our case we use the MDGs since the data is represented in terms of abstract variables and the operations are represented in terms of uninterpreted function symbols.

In this thesis, we concentrate on behavioral description written in MDG-HDL language. The authors in [25] were able to automatically generate a formal model from the design description. We were able to extract a DF model from the behavioral design (subset of VHDL or SystemC) based on some rules explained in [25, 58]. The following steps summarize partially the extraction method:

1. First, we translate all simple assignment expressions (equations) to DF. The variable and expression propagation and algebraic simplification is done as: for any expression we need to propagate the variables and apply a simplification on them. The simplification includes Boolean algebraic where True and False can be removed. The example below explain this case:

$$a = b + 1; b = 2 * a; \Longrightarrow \begin{cases} a = b + 1; \\ b = 2 * (b + 1) \end{cases}$$

2. In the second step, the set of sequential assignment is translated to a set of conjunct equations: $eq1; eq2; \Longrightarrow eq1 \wedge eq2$.

3. In the third step, we handle if-then-else statement by the following steps:

(a) Variable and assignment expression: as described above.

(b) Every statement written in the form $x = if(condition, then\_branch, else\_branch)$ will be translated into a set of equalities of form: $(condition \wedge (x = then\_branch)) \vee (\neg condition \wedge (x = else\_branch))$, where the condition ($cond$) is of concrete type (has enumeration).

4. In the fourth step, the uninterpreted functions are defined whenever it is necessary. In this step, we translate any function between the operators in the right hand side of the equation to uninterpreted functions representation. For example, $x' = x + 1$ become $x' = plus(x, 1)$. Also, for the bit vector and word level we expressed them as uninterpreted function symbols (abstraction).

5. Finally, any other synthesizable statements and much more elements such as multiple clocks and unbounded integer are intended for system level modeling can be translated to if-then-else statements as presented in [25, 58].

Just as an example, consider a basic RAM element with two operations write and read (read one word or two words) defined at behavior level:

Inputs:{reset, addA, addB, data, reg_one,write, read_one, output_enable}

Outputs: {outputA, outputB}

Registers: $\{reg[0] \ldots reg[n]\}$

Writing Process

if reset=1 **then**

reg[addA]=0

reg[addB]=0

else if (write=1) and (output_enable=0) **then**

reg[addA]=data

end if

Reading Process

if (write=0) and (read_one=1) and (output_enable=1) **then**

(outputA=reg[addA])

else if (write=0) and (read_one=0) and (output_enable=1) **then**

(outputA=reg[addA])

(outputB=reg[addB])

end if

The transition relations for the write operation in terms of DF $Write\_Tr_{DF}$ is written as follows:

$$[(reset = 1) \wedge (reg[addA] = 0) \wedge (reg[addB] = 0)] \bigvee$$
$$[(reset = 0) \wedge (write = 1) \wedge$$
$$(output\_enable = 0) \wedge (reg[addA] = data)]$$

and for the read operation $Read\_Tr_{DF}$:

$$[(write = 0) \wedge (read\_one = 1) \wedge$$
$$(output\_enable = 1) \wedge (outputA = reg[addA])] \bigvee$$
$$[(write = 0) \wedge (read\_one = 0) \wedge (output\_enable = 1) \wedge$$
$$(outputA = reg[addA]) \wedge (outputB = reg[addB])]$$

The transition relation for the design is the conjunction of both transition relations of the read and the write operations, in DF that will be written as follows:

$$RAM\_Tr_{DF} = Read\_Tr_{DF} \wedge Write\_Tr_{DF}$$

From the Properties

In large systems where the design can be expressed as a conjunction of individual transition relations of the state variables, it consumes large memory and time to verify a property. If the property to be verified is only affected by a part of the system behavior, we can use the corresponding subset of the transition relations to verify the property. In order to enhance the reduction of the design prior to verification, we use the precondition in the property (antecedent) and express them in terms of DF. Note that the property is written in $\mathcal{L}_{MDG}$ and in the form of $[Ante \rightarrow Cons]$, where both *Ante* and *Cons* are directed formulae called *antecedent* and *consequent*.

Lets back to our previous RAM example, if we want to verify a property about the writing operation of the memory then the property can be expressed as $AG((write = 1) ==> X(reg[addA] = data))$ and its DF will be $P\_Write\_ant = (write = 1)$. This DF will be then used to reduce the RAM_Tr_{DF}.

6.3.3 Verification of the Reduction Soundness

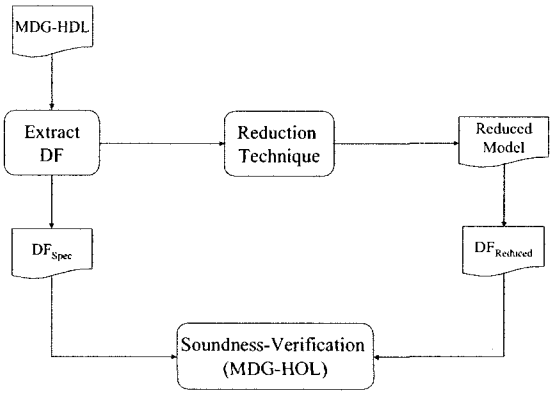


Figure 6.5: Overview of the Soundness-Verification Methodology

As shown in Figure 6.5, we start with a specification of a circuit design written

in Hardware Description Language (MDG-HDL) and extract a mathematical model in terms of Directed Formulae (DF_{Spec}). After applying the MDG-HOL reduction technique on the DF of the design, the reduced model is generated and expressed in terms of Directed Formulae ($DF_{Reduced}$).

Then, both DFs should be fed to the MDG-HOL platform where the soundness verification is checked. If the reduction is proved sound then the formal verification can be performed on the obtained reduced model.

The powerful of our methodology is that it can be used with any verification tool. All what we need is to translate in a sound manner both the model and its reduction in order to embed them thereafter as DFs in HOL and then prove that the reduced model is derived correctly using high order logic.

Next, we present an algorithm to achieve the verification. In verification world, the design transition relation Tr should satisfy the specification. Intuitively, we want Tr to be a specification-consistent (not spec-contradictory) for any input and state combinations. In fact, spec-consistency does not depend on any upper level specification and one can, in some sense, view the transition relation at high level as the specification (facts can be mapped into mathematical statements within the specification). Thus, we can determine whether a design is a spec-consistent without having a reference specification[70]. The section requires that Tr and $\{\varphi_i\}_{1\leq i\leq n, j\neq i}$ are embedded in HOL as Directed Formulae.

Definition 6.3.1 *The spec-consistency*

We suppose that the specification can be written as a set of n properties $\{\varphi_i\}_{1\leq i\leq n}$. *The spec-consistency of* Tr *can be defined as:*

$$\forall j, 1 \leq j \leq n, Tr \models \{\varphi_i\}_{1\leq i\leq n, j\neq i} \Rightarrow Tr \models \varphi_j$$

The Reduction-Soundness Algorithm

In fact, any reduction can be considered as a partial spec. Thus, if the reduction technique is sound then the specification of the reduced system should be consistent with the original one. Then, in order to verify that Tr satisfies φ_j, we assume that Tr satisfies all other properties $\{\varphi_i\}_{1\leq i\leq n, j\neq i}$. In this context, the following definition describes the reduction soundness:

Definition 6.3.2 *The reduction soundness*
Let M and M' be a two ASM models. We say that M' is soundly reduced model: $M' \preceq M$ if and only if:

- *for any property P such that: $M' \models P$ then P holds in the original model M: $M \models P$.*

Our algorithm not only verifies the soundness of the reduction but also can determine a minimum set of sound property clauses (equalities). The algorithm is operating on the reduced DFs. We need one DF for Tr of the spec or the design under verification (DF_{spec}) and a set of DFs for each property clause (DF_{P_i}), where DF_{P_i} represents the antecedent of the property. As shown in Algorithm 8, lines 1 and 2 store the initial DFs. The variables ϕ and φ denote the DF of the original spec and the DF of the property clause, respectively. Lines 3-10 repeatedly execute a loop n times, where n still represents the number of property clauses. Line 4 computes the reduction step (Section 4.1) by evaluating the conjunction operation and then applying the propagation of the property clause as rewriting rule. The soundness of the reduction step is tested in line 5 by using the prune by subsumption operation (PbyS). If $(PbyS(\phi_i, DF_{spec}) = F)$ then the behavior of the reduced model is included in the original model and thus, the reduction property is guaranteed to be correct

as shown in line 6. Otherwise, this property is removed from the properties clauses which mean that the system is not reduced without influencing the behavior (over reduction). This property will not be used in the reduction process. The algorithm returns the guarantee clauses as shown in line 11.

Algorithm 8 SOUND_REDUCE_DF($\{DF_{spec}\}$, $\{DF_{P_i}\}_{0<i\leq n}$)

1: $\phi_0 = DF_{spec}$;
2: $\varphi_0 = \{\}$;
3: **for** $i = 1$ to n **do**
4: $\quad \phi_i = Reduce(\phi_0, DF_{P_i})$;
5: $\quad$ **if** $(PbyS(\phi_i, DF_{spec}) = F)$ **then**
6: $\quad\quad \varphi_i = \varphi_{i-1} \bigcup DF_{P_i}$;
7: $\quad$ **else**
8: $\quad\quad \varphi_i = \varphi_{i-1}$;
9: $\quad$ **end if**
10: **end for**
11: $Guarantee\_P = \varphi_i$;

The algorithm is correct since it returns a unique set for the same inputs. Also, the algorithm terminates because we have a finite number of clauses executed n times in the loop.

Correctness of the Algorithm

Two theorems are stated to prove the correctness of the algorithm. The first one describes the correctness proof of the MDG operations which represents a mathematical proof of consistency between the operation specification and its implementation in HOL as explained in Section 4.5.

The second theorem regarding that the PbyS itself is a checking for the soundness of the reduction technique; guarantees that the reduced model is included in the original model for the same property. This means if the property holds for both the reduced and original model then the reduction is correct for the same property P: if

$\phi' \models P$ then $\phi \models P$.

Theorem 6.3.1 *Pbys checks the Reduction Soundness*

ASSUME:

1. M and M' be a two ASM models: $M' \preceq M$.

2. DF_{spec} and Reduced_DF be the respective transition relation in terms of well-formed DF of M and M'.

Then the reduction approach is sound if:

$$PbyS(Reduced\_DF, DF_{spec}) = F$$

PROOF:

Since *Reduced_DF* represents the transition relation of the model M' which should be included in M, the *Reduced_DF* formula cannot be a T or F (see definition 6.3.2). The only interesting case is when *Reduced_DF* is not T or F. By applying the definition of PbyS as shown in (4.4), the result R is derivable from *Reduced_DF* by pruning. Hence $\models R \Rightarrow Reduced\_DF$. And, from (4.4), it follows tautologically that $\models$Reduced_DF$\wedge\neg(\exists E)DF_D \Rightarrow R$. Thus we have

$$\models (Reduced\_DF \wedge \neg(\exists E)DF_D \Rightarrow R) \wedge (R \Rightarrow Reduced\_DF)$$

which holds if and only if R is F, then it follows tautologically from (4.4) that $\models$ $Reduced\_DF \Rightarrow (\exists E)DF_D$. We have thus proved the soundness the reduction. □

The False Negative

Our verification algorithm is not complete as it can provide false negative results. That means a correct reduction may not be proved to be sound. The situation can be compared to abstraction when data operations are viewed as black boxes, then the invariant is expected to hold for every interpretation. Hence, if the proof returns

failure, then there must be an interpretation that gives the expected error. However, in the concrete model this interpretation is not achievable.

Definition 6.3.3 *False Negative*

Let M and M' be a two ASM models and M' is a soundly reduced model: $M' \preceq M$. Then the false negative occurs when a property P dose not hold on M': $P \not\models M'$ but holds in the original model M: $M \models P$.

Thus, our method can prove that the reduction technique is soundly applied but cannot prove the opposite. This due to the over-approximation nature of our generalized algorithm. For some special cases, we can prove the absence of these false negative depending on the applied reduction technique. However, this proof may not be automatic and needs an intervention from an ATP expert.

The RAM Example

If we want to verify a property to read from the RAM, then the property will include $(write = 0)$. Then by using the MDGs operations and rewriting engine of HOL, the transition relations of the write operations will be eliminated from the design DF. Also, the *reset* input is eliminated by expressing this as a property: activating it once then always remains deactivated since the interesting properties does not include *reset*.

Lets have the following property to read one word from the memory then the property can be expressed as: $AG((write = 0 \ \& \ read\_one = 1) ==> X(outputA = reg[addA]))$ and its antecedent will be $P\_read\_one\_ant = (write = 0)\&(read\_one = 1)$. In this case, the new reduced RAM_Tr_{DF} will be:

$[(write = 0) \wedge (read\_one = 1) \wedge$

$(output\_enable = 1) \wedge (outputA = reg[addA])]$

Then by applying Algorithm 8, it is easy to prove that the resulted DF is included in the original DF design. Finally, the reduced DF with the property can be mapped to the model checker. Thus, an important reduction is gained on the resulting transition relation which improves the performance of the model checker in terms of memory and CPU time.

On the other hand, if we have a case such that the resulted DF is not included in the original DF design, then this may be regarding an improper property or a problem in the design itself. For example, lets have a property expressed as $AG((write\_one = 0) ==> X(outputA = reg[addA]))$. In this case, the antecedent of the property will not reduce the original design since the design has not a variable *write_one*. Using Algorithm 8, the PbyS operation will give a result other than F. Then, this property will be removed from the property clauses under the condition that our specification is consistent.

6.3.4 Case Studies

We have verified the reduction of the assume-guarantee method with two circuits: the Look-Aside Interface and the Island Tunnel Controller in order to measure the performance of our approach. The results were carried out on a Sun enterprize server with Solaris 5.7 OS and 6.0 GB memory (refer to [5] for more details on each application).

Look-Aside Interface (LA-1)

System Description

The LA-1 interface [29], developed by the Network Processor Forum (NPF), is a memory mapped interface based on Quad Data Rate (QDR) and Sigma RAM technologies (SRAM). It targets look-up-tables and memory-based coprocessors and emphasizes as much as possible on the use of the existing technology. The LA-1 specification aims to accommodate other devices as well, such as classifiers and encryption co-processors. The major features of the LA-1 interface include:

- Concurrent read and write operation.
- Separate unidirectional read and write data buses.
- Single address bus.
- 18-bit DDR data output bus transfers 32 bits plus 4 bits of even parity per read.
- 18-bit DDR data input bus transfers 32 bits plus 4 bits of even parity per write.
- Byte write control for writes.

The LA-1 interface transfers data between an Network Processor Unit (NPU) and memory or coprocessors. Figure 6.6 shows the LA-1 interface bus signals. LA-1 requires a master-clock pair. The master clocks (K and $K\#$) are ideally 180 degrees out of phase with each other, and they are outputs for the host device and inputs for the slave device. A write cycle is initiated by asserting WRITE SEL ($W\#$) low at rising edge of K (K clock). The address of the Write cycle is provided at the following edge of K ($K\#$ clock which 180 degrees out phase from clock K). A read cycle is initiated by asserting READ SEL ($R\#$) low at rising edge of K (K clock) and the read address is presented on the same rising edge. There is also 2-bit active-low byte-write inputs ($BW\#$) and a 16-bit synchronous data inputs (D) plus 2-bit synchronous data even inputs (DP) for write operations. Similarly, it has a 16-bit

synchronous data outputs (DO) plus 2-bit synchronous data even outputs (DPO) for reads.

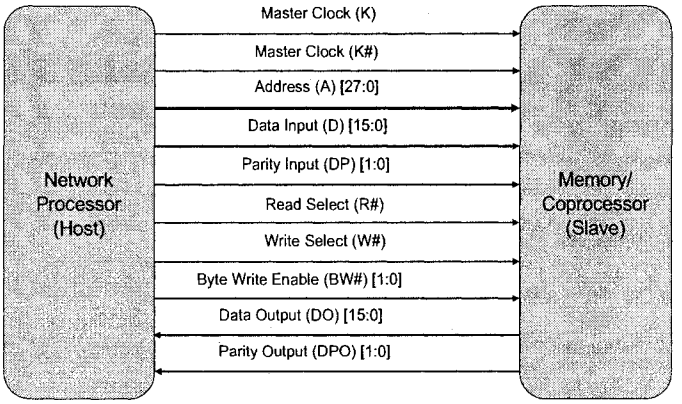


Figure 6.6: Look-Aside Interface

The MDG-HDL model for the LA-1 design is shown in Figure 6.7, where:

- input signals: K, $K\#$, $W\#$, $R\#$, *DP0*, *DP1*, *BW0*, *BW1* and *pflag* are of type *bool*,
- input signals: D, A and d_m are of abstract sort *wordn*,
- output signals: *me*, *bwe_m3*, *bwe_m2*, *bwe_m1*, *bwe_m0*, *DPO0* and *DPO1* are of type *bool*,
- output signals: *d2m*, *add_r* and DO are of abstract sort *wordn*, and
- components: *make_word*, *parity4*, *parity3*, *parity2*, *parity1*, *msw* and *lsw* are abstract function symbols.

Note that an internal double frequency clock is used to generate the *clock_2X* and the control signal *pflag* is used to indicate the positive and negative edge of the clock. The function of *make_word* is to merge two input data into one output data.

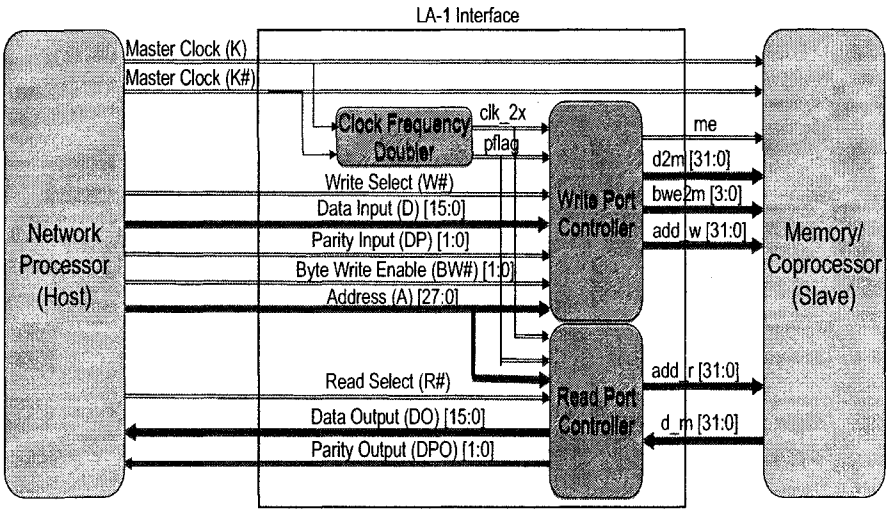


Figure 6.7: Look-Aside Interface Design

The function of *parity4*, *parity3*, *parity2* and *parity1* is to compute the parity of the input data. The function of *msw* and *lsw* is to strip the most and least significant word from the input data, respectively.

Verification In following, we describe our results on the verification of the LA-1 Interface using the MDG-HOL based reduction technique over some properties. We describe four properties that we extracted and verified from the design specifications:

- **Property 1** (Write Port): by asserting $W\#$ low at the rising edge of K, the active-low memory enable signal *me* will be set to low at the next rising edge of K:

  ```
  AG( (pflag=1 & W=0) ==> (XX (me=1)) );
  ```

- **Property 2**(Write Port): by asserting $W\#$ low at the rising edge of K, if the

byte-write control inputs *BW#1* and *BW0#* are set to low, the full data input D will be written at the same cycle and at the rising edge of K and $K\#$ and sent to the memory through *dm* (data to memory). This scenario is known to be the pass through mode of the Write Port:

```
AG( (pflag=1 & W=0 & BW1=0 & BW0=0) ==>
        (LET (v1=D) IN
          (X (LET (v2=D) IN
              (X (dm = fmake_word(v1,v2)))))) );
```

- **Property 3** (Read Port): by asserting $R\#$ low at the rising edge of K, the data from the memory *d_m* will be sent through data even output *DPO(1)* after the next rising edge of K:

```
AG( (pflag=1 & R=0)==>
       (XX (LET (v1=d_m) IN
           (XX (DPO1=fparity2(v1)) ))) );
```

- **Property 4** (Read Port): by asserting $R\#$ low at the rising edge of K, the data from the memory *d_m* will be sent through DO after the next rising edge of K:

```
AG( (pflag=1 & R=0) ==>
        (XX (LET (v1=d_m) IN
          (X ( (DO=fmsw(v1))
              & (X (DO = flsw(v1)))))) );
```

Table 6.2 compares the verification results of the original MDG model checking and the reduced one with soundness verification for four properties. The reduction time includes the verification of the reduction soundness. The CPU time is measured

Table 6.2: Comparing the Original MDGs Model Checking Results with the Reduced MC and Soundness Verification Results

Benchmark	*Original MC*			*Reduced MC*		
Properties	**Time**	**Mem**	**Nodes**	**Time**	**Mem**	**Nodes**
P1	435.56	449.7	1273027	48.90	59.6	168060
P2	63.82	70.6	217043	1.32	2.84	4630
P3	233.47	248.6	716544	46.05	52.7	172719
P4	245.21	269.1	765434	36.62	51.7	111864
Average	244.51	259.5	743012	33.22	41.71	114318

in seconds and the memory is measured in MB.

The first two properties are used to verify the write port while the last two properties are used to verify the read port. The best gain in performance is obtained with property P2 where the time is reduced by 48 times the original one and the memory is reduced by a factor of 24.8 times. The worst case is the property P3 where the time is reduced by 5 times the original one and the memory reduction is not profitable. We note that the reduction gain depends on the properties. Also, the read port circuit is bigger than the write port circuit and hence, the first two properties took much less time compared to the last two.

Note that the soundness verification using the PbyS operation took less than 7 seconds added to the average time. These differences show the sensitivity of the reduction technique to the property verified. Despite these fluctuations, the average of the gain in performance is a factor of 6.1 which is considered as a good result in the case of model checking approaches.

Island Tunnel Controller (ITC)

Verification

Table 6.3: Comparing the Original MDGs Model Checking Results with the Reduced MC and Soundness Verification Results

Benchmark	*Original MC*			*Reduced MC*		
Properties	**Time**	**Mem**	**Nodes**	**Time**	**Mem**	**Nodes**
P1	121.12	100.91	226070	17.05	10.92	26787
P2	65.26	48.6	123080	0.03	0.22	44
P3	81.73	58.2	171060	15.84	10.24	22278
P4	67.07	48.8	123080	9.28	6.88	20125
P5	66.08	49.4	123080	12.09	8.01	20278
Average	80.25	61.18	153274	10.86	7.254	17902

In following, we describe our results on the verification of the ITC. We have specified and verified a number of properties on the *ITC*. In the following, we describe five samples for illustration purposes:

- **Property 1**: The cars at the island entrance will enterally pass the tunnel.
- **Property 2**: The green light of *ILC* must be off if there is a car exiting the tunnel.
- **Property 3**: The island will eventually release the control right of the tunnel controller requests.
- **Property 4**: The tunnel counter keeps the old value if ordered to increment and decrement at the same time.
- **Property 5**: The green light of MLC must be on if there is no request to yield control of the tunnel and the number of cars on the island are less than n.

Table 6.3 compares the verification results of the original MDG model checking and the reduced one with soundness verification for seven properties. We give the CPU

time measured in seconds and the memory measured in MB that are used in building the reduced machine and checking the property. The best gain in performance is obtained with property P2 where the time is reduced by 2175 times the original one and the memory is reduced by a factor of 220 times. The worst case is the property P3 where the reduction in time and memory is 5 times the original one. In the case of property P2 the assumptions includes two global signals that causes a huge reduction on the complete transition relation which really was much more efficient. For P3, it was only one local signal in the assumption of the property which results a small impact on the global transition relation.

The soundness verification using the PbyS operation took less than 10 seconds added to the average time. The average of the gain in performance is a factor of 4.5 which is considered as a good result in the case of model checking approaches. Still, we have verified for each case the soundness of the verification. Moreover, the verification time and the soundness reduction time is still less than the model checking verification time.

Chapter 7

Conclusions and Future Work

7.1 Summary

In this thesis, we have proposed a high level reachability approach using the MDG syntax and embedded operations using the HOL theorem prover. We have provided a link from the MDG graphs to formulae in high order logic using the Directed Formulae notations. Afterward, we have defined a tactic to check the satisfaction of the well-formedness conditions of these Directed Formulae. In order to follow the formal logic of HOL, the formalization of the Directed Formulae in [88] has been modified. Therefore, the modified DF formalization is more suitable for automatic reasoning and is helpful for avoiding potential infinite loops. Moreover, it ensures the termination when it should occur [6]. In fact, applying induction on *DF*, with these modifications, ameliorate the reasoning with the MDG structure in HOL. This is one of the contributions of our work.

The formalization of MDG operations is built on 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 proofs. The verification was conducted using the

deep embedding approach, which ensures the consistency of our approach. Since we do everything in HOL, we expect higher security than other implementations in high level languages such as C. Also, 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.

We have also presented some experimental results based on four benchmarks. From these experiments, combined with abstract sorts and uninterpreted functions, 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 the Formalcheck. The idea here is not to compete with the traditional model checkers but to show the performance of using our platform as well as the possibility of future integration.

We have proposed a reduction technique for MDG model checking based on SAT and MDG-HOL integrated platform. Also, we have proposed a method to verify that the reduction techniques are applied soundly. The benefit of our approach is that it can be applied within any verification system to produce a sound reduced systems without major penalty over verification performance. The specification of the design described at system level language along with properties are used to verify the reduced model. The originality of our technique comes from combining an automatic technique (MDGs operations) and a non-automatic tool (HOL) to prove in High Order Logic that the reduced model is derived correctly in an automatic manner.

We support our technique by experimental results executed on benchmark properties. The obtained performance in the case of SAT-MDG is acceptable as compared with commercial model checking tools. Even if bit-blasting tools can perform ten

times faster, our technique is safer as it provides proof that the results are derived correctly. In the case of assume guarantee in MDG-HOL, the reduction strategy was limited to the propagation of antecedents of the properties. The obtained results still satisfactory.

In fact, our approach can be used to express more reduction techniques without any loss of generality, without loss of automatism, and more importantly, automatic soundness checking.

7.2 Future Research Directions

The work presented in this thesis is an important step, to define an algorithm for states exploration inside an inductive theorem prover; forward to tackle higher level of abstraction. Based on our previous work in this domain, we believe that the proceeding future work can be completed and expanded in the following manner:

- **MDG-HOL platform**:

    1. Improving the performance of the MDG-HOL component by improving the existing code and adding standard optimizations.

    2. The work can be extended to implement a complete high level model checking in HOL based on our infrastructure. Including the definition of each $\mathcal{L}_{MDG}$ [88] related algorithm as a tactic. 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 tactic can be used to make calls to our defined MDG algorithms, to check whether an L_{MDG} property is valid. Here, we are not reducing the role of the proof expert, but we provide him with an automated tactics 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. Also, building a parser to automatically extract the transition relation from HDL is another future step that we intend to do using ML.

3. Performance is an important issue to convince industrial practitioners who are usually interested in absolute performance figures. As is evident from Chapters 5 and 6, much of current research is focused on enabling cooperation between various techniques. Though in theory we can implement any such technique in HOL, the absence of a general framework gives any implementation an ad hoc nature. All we have is a philosophy: do everything fully-expansively for better assurance of soundness, closer integration and exploit the asymmetric cost of proof checking vs. proof search whenever possible for efficiency. Though, we have taken first steps, MDG-HOL framework that embodies this philosophy with an emphasis on combining technologies would be desirable. Whether or not this would be over-engineering depends on the eventual domain of use.

4. The approach of embedding MDG algorithms in a theorem prover carries a clear performance penalty. There are important issues about what kind of problems this approach is best suited to. There are also engineering issues about how to improve the performance without abandoning the fully-expansive approach such as using the code generation of HOL to generate an ML code for the MDG-HOL platform and then embed the MDG model checking algorithm as a shallow embedding. This can be done by the invocation of `emitML` to generate an ML signature and structure files. And

finally, there is usage issue about just what is considered an acceptable penalty in a given situation. All these needed to be addressed at some point.

- **Reduction Techniques**:

1. Our approach can be easily generalized to any other verification tools such as commercial model checker (RuleBase) or SAT solvers. In this case, we need to build a parser to translate the reduced model to DF. Ideally, the reduction technique itself should be formalized in HOL on order to be verified with our method.
2. One of the limitations of our approach is that we have a one way soundness proof (not bisimulation). Also the output of the reduction algorithm is a directed formula. Those limitations do not contradict the fact that we have better results as well as highlight some of the limitations to improve our approach. More work is needed to resolve those limitations.
3. Future directions will concentrate on embedding more reduction techniques inside MDG-HOL platform. The most important, will be the embedding of SAT solvers advanced heuristics as the one used with miniSAT [60] and RSAT [41].

Bibliography

[1] M. Aagaard, R. B. Jones, R. Kaivola, K. R. Kohatsu, and C. H. Seger. Formal Verification of Iterative Algorithms in Microprocessors. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 201–206, New York, NY, USA, 2000. ACM.

[2] M. D. Aagaard, R. B. Jones, and C. H. Seger. Combining Theorem Proving and Trajectory Evaluation in an Industrial Environment. In *Proceedings of the 1998 Conference on Design Automation (DAC-98)*, pages 538–541, Los Alamitos, CA, June 1998. ACM/IEEE.

[3] P. Aziz Abdulla, P. Bjesse, and N. Eén. Symbolic Reachability Analysis based on SAT-Solvers. In *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2000)*, 2000.

[4] S. Abed, O. Ait Mohamed, and G. Al Sammane. On the Embedding and Verification of Multiway Decision Graphs in HOL Theorem Prover. Technical Report 2007–1–Abed, ECE Department, Concordia University, Montreal, Canada, February 2007.

[5] S. Abed, O. Ait Mohamed, and G. Al Sammane. HOL based Reduction Techniques for MDGs Model Checking. Technical Report 2008–1–Abed, ECE Department, Concordia University, Montreal, Canada, January 2008.

[6] 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.

[7] H. 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.

[8] M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley, editors. *A Simple Graph Theory and Its Application in Railway Signalling.* IEEE Computer Society, 1992.

[9] R. J. Bayardo and R. C. Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT Instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 203–208, Providence, Rhode Island, 1997.

[10] S. Bensalem, Y. Lakhnech, and S. Owre. InVeSt: A Tool for the Verification of Invariants. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer-Aided Verification, CAV '98*, volume 1427, pages 505–510, Vancouver, Canada, 1998. Springer-Verlag.

[11] P. Bjesse and K. Claessen. SAT-based Verification without State Space Traversal. In *Formal Methods in Computer-Aided Design*, pages 372–389, 2000.

[12] R. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. Brady. Deciding Bit-Vector Arithmetic with Abstraction. In *Proc. TACAS 2007*, March 2007.

[13] R. E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[14] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[15] C.-T. Chou. A Formal Theory of Undirected Graphs in Higher-Order Logic. In T.F. Melham and J. Camilleri, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 859, pages 144–157, Malta, 1994. Springer-Verlag.

[16] C.-T. Chou. Mechanical Verification of Distributed Algorithms in Higher-Order Logic. In T.F. Melham and J. Camilleri, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 859, pages 158–176, Malta, 1994. Springer-Verlag.

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

[18] L. Cai and D. Gajski. Transaction Level Modeling: An Overview. *Hardware/Software Codesign and System Synthesis, 2003. First IEEE/ACM/IFIP International Conference on*, pages 19–24, 1-3 Oct. 2003.

[19] E. M. Clarke, O. Grumberg, and D. E. Long. Model Checking. In *Nato ASI*, volume 152 of F. Springer-Verlag, 1996.

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

[21] 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.

[22] O. Coudert, C. Berthet, and J. C. Madre. Verification of Synchrounous Sequential Machines based on Symbolic Execution. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[23] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. *A Tutorial Introduction to PVS*. http://www.dcs.gla.ac.uk/proper/papers.html.

[24] D. Long. *Model Checking, Abstraction, and Compositional Verification*. PhD thesis, Pittsburgh, PA, USA, 1993.

[25] D. Toma, D. Borrione, and G. Al Sammane. Combining Several Paradigms for Circuit Validation and Verification. In *CASSIS, Selected Papers, LNCS*, volume 3362/2005, pages 229–249, Marseille, France, 2004. Springer-Verlag GmbH.

[26] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-Proving. *Commun. ACM*, 5(7):394–397, 1962.

[27] F. Emerson and A. Sistla. Symmetry and Model Checking. *Formal Methods in System Design*, 9(1/2):105–131, August 1996.

[28] K. Fisler and K. Johnson. Integrating Design and Verification Environments Through A Logic Supporting Hardware Daigrams. In *Proc. IFIP Conference on Hardware Description Languages and their Applications (CHDL'95)*, Chiba,Japan, August 1995.

[29] Network Processing Forum. Look-Aside (LA–1) Interface, Implementation Agreement, Revision 1.1, April 15 2004.

[30] M. K. Ganai and A. Aziz. Improved SAT-based Bounded Reachability Analysis. In *ASP-DAC '02: Proceedings of the 2002 conference on Asia South Pacific design automation/VLSI Design*, page 729, Washington, DC, USA, 2002. IEEE Computer Society.

[31] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[32] M. Gordon. From LCF to HOL: A Short History. pages 169–185, 2000.

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

[34] 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.

[35] M. J. C. Gordon. Programming Combinations of Deduction and BDD-based Symbolic Calculation. *LMS Journal of Computation and Mathematics*, 5:56–76, August 2002.

[36] 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.

[37] J. Harrison. Binary Decision Diagrams as a HOL Derived Rule. *The Computer Journal*, 38:162–170, 1995.

[38] J. Harrison. Towards Self-Verification of HOL Light. In Ulrich Furbach and Natarajan Shankar, editors, *Proceedings of the third International Joint Conference, IJCAR 2006*, volume 4130, pages 177–191, Seattle, WA, 2006.

[39] S. Hazelhurst, O. Weissberg, G. Kamhi, and L. Fix. A Hybrid Verification Approach: Getting deep into the Design. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 111–116, New York, NY, USA, 2002. ACM.

[40] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1990.

[41] SAT05 Competition Homepage. http://www.satcompetition.org/2005/.

[42] G. Huet, G. Kahn, and C. Paulin-Mohring. *The Coq Proof Assistant : A Tutorial.* http://coq.inria.fr/doc/tutorial.html.

[43] J. Hou and E. Cerny. Model Reductions in MDG-based Model Checking. In *13th Annual IEEE International ASIC/SOC Conference*, pages 347–351. IEEE, 2000.

[44] J. J. Joyce and C. H. Seger, editors. *The HOL-Voss System: Model Checking inside a General-Purpose Theorem Prover*, volume 780 of *Lecture Notes in Computer Science.* Springer, 1994.

[45] K. McMillan. Interpolation and SAT-based Model Checking. In Warren A., Hunt Jr. and Fabio Somenzi, editor, *Proceedings of the International Conference On Computer Aided Verification*, volume 2725, pages 1–13. Springer Verlag, 2003.

[46] K. Schneider and T. Kropf. Verifying Hardware Correctness by Combining Theorem Proving and Model Checking. Technical Report SFB358-C2-5/95, Institut für Rechnerentwurf und Fehlertoleranz, 1995.

[47] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, June 2000.

[48] 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.

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

[50] S. Krstic and J. Matthews. Verifying BDD Algorithms through Monadic Interpretation. In *VMCAI '02: Revised Papers from the Third International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 182–195, London, UK, 2002. Springer-Verlag.

[51] R. P. Kurshan and L. Lamport. Verification of a Multiplier: 64 Bits and Beyond. In C. Courcoubetis, editor, *Proceedings of the 5th International Conference on Computer Aided Verification*, volume 697, pages 166–179, Elounda, Greece, 1993. Springer-Verlag.

[52] J. P. Marques-Silva and K. A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 220–227, 1996.

[53] K. McMillan. Verification of Infinite State Systems by Compositional Model Checking. In *CHARME '99: Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 219–234, London, UK, 1999. Springer-Verlag.

[54] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Boston, Massachusetts, 1993.

[55] T. Melham. Integrating Model Checking and Theorem Proving in a Reflective Functional Language. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Integrated Formal Methods: 4th International Conference, IFM 2004: Canterbury, UK, April 4–7, 2004: Proceedings*, volume 2999 of *Lecture Notes in Computer Science*, pages 36–39. Springer-Verlag, 2004.

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

[57] R. Mizouni, S. Tahar, and P. Curzon. Hybrid Verification Incorporating HOL Theorem Proving and MDG Model Checking. *Microelectronics Journal*, 2006.

[58] J. Moore. Introduction to the OBDD Algorithm for the ATP Community. *Journal of Automated Reasoning*, 12(1):33–46, 1994.

[59] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.

[60] N. Sorensson and N. Een. MiniSat v1.13 A SAT Solver with Conflict-Clause Minimizationn. In *Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT 2005)*, volume 3569, St. Andrews, UK, 2005. Springer-Verlag.

[61] K. Namjoshi and R. Kurshan. Syntactic Program Transformations for Automatic Abstraction. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 435–449, London, UK, 2000. Springer-Verlag.

[62] K. S. Namjoshi. Certifying Model Checkers. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 2–13, London, UK, 2001. Springer-Verlag.

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

[64] V. Ortner and N. Schirmer. Verification of BDD Normalization. In *TPHOLs*, pages 261–277, 2005.

[65] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, New York, NY, USA, 1991.

[66] L. C. Paulson. *Isabelle: A Generic Theorem Prover.* Springer Verlag, 1994.

[67] V. Pisini, S. Tahar, P. Curzon, O. Ait-Mohamed, and X. Song. Formal Hardware Verification by Integrating HOL and MDG. In *Proc. of IEEE GLS-VLSI'00, Chicago, USA*, Chicago, Illinois, USA, 2000.

[68] J. Quille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In M. Dezani-Ciancaglini and Ogo Montanari, editors, *Proceedings of the 5th International Symposium on Programming*, volume 137, pages 337–351. Springer-Verlag, 1982.

[69] R. Boulton, A. Gordon, M.J.C. Gordon, J. Herbert, and J. van Tassel. Experience with Embedding Hardware Description Languages in HOL. In *Proc. of the International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 129–156, Nijmegen, 1992. North-Holland.

[70] R. Jones, C. -J. Seger, and D. Dill. Self-Consistency Checking. In M. Srivas and A. Camilleri, editors, *First international conference on formal methods in*

computer-aided design, volume 1166, pages 159–171, Palo Alto, CA, USA, 1996. Springer-Verlag.

[71] W. Reif, J. Ruf, G. Schellhorn, and T. Vollmer. Do you Trust your Model Checker? In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Formal Methods in Computer Aided Design (FMCAD)*. Springer LNCS 1954, November 2000.

[72] T. Ridge. Graphs and Trees in Isabelle/HOL. Technical report, 2005.

[73] S. Graf and H. Saidi. Verifying Invariants using Theorem Proving. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 196–207, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[74] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 411–414, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[75] S. Rajan, N. Shankar, and M. K. Srivas. An Integration of Model Checking with Automated Proof Checking. In P. Wolper, editor, *Proceedings of the 7th International Conference On Computer Aided Verification*, volume 939, pages 84–97, Liege, Belgium, 1995. Springer Verlag.

[76] G. Al Sammane. *Symbolic Simulation of Circuits Described at Algorithmic Level.* PhD thesis, Joseph Fourier University, 2005.

[77] G. Al Sammane, S. Abed, and O. Ait Mohamed. High Level Reduction Technique for Multiway Decision Graphs based Model Checking. In *First International*

Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS 2007), Algiers, Algeria, May 2007. The British Computer Society.

[78] K. Schneider and D. Hoffmann. A HOL Conversion for Translating Linear Time Temporal Logic to ω-automata. In *TPHOLs*, volume 1690, pages 255–272, Nice, France, 1999. Springer-Verlag.

[79] C. H. Seger. Voss – A Formal Hardware Verification System, User's Guide. Technical report, Nortel Networks, Ottawa, Canada, The University of British Columbia, December 1993. ftp ftp.cs.ubc.ca:ftplocaltechreports1993TR-93-45.ps.

[80] S.Hazelhurst and C.-J.H. Seeger. Symbolic Trajectory Evaluation. In *Formal Hardware Verification - Methods and Systems in Comparison*, pages 3–78. Springer-Verlag, London, UK, 1997.

[81] M. Sheeran, S. Singh, and G. Staalmarck. Checking Safety Properties using Induction and a SAT-Solver. In *FMCAD '00: Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pages 108–125, London, UK, 2000. Springer-Verlag.

[82] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. In *7th. Symposium of Logics in Computer Science*, pages 394–406, Santa-Cruz, California, 1992. IEEE Computer Scienty Press.

[83] A. Valmari. A Stubborn Attack on State Explosion. In *CAV '90: Proceedings of the 2nd International Workshop on Computer Aided Verification*, pages 156–165, London, UK, 1991. Springer-Verlag.

[84] M. Velev. Using Automatic Case Splits and Efficient CNF Translation to Guide a SAT-Solver when Formally Verifying Out-of-Order Processors. In *AMAI*, 2004.

[85] 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.

[86] J. Wright. Representing Higher-Order Logic Proofs in HOL. In *Proceedings of the 7th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, pages 456–470, London, UK, 1994. Springer-Verlag.

[87] H. Xiong, P. Curzon, S. Tahar, and A. Blandford. Providing a Formal Linkage between MDG and HOL. *Formal Methods in Systems Design*, 29(3):1–36, 2006.

[88] 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.

[89] H. Zhang. SATO: An Efficient Propositional Prover. In *Proceedings of the International Conference on Automated Deduction (CADE'97), volume 1249 of LNAI*, pages 272–275, 1997.

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

[91] 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 *FMCAD '96: Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, pages 233–247, London, UK, 1996. Springer-Verlag.

Appendix A

The MDG-HOL Platform

This appendix includes the formalization details and proofs of the MDG-HOL platform (data structure + operations). It is used for illustrating purpose. The complete embedding of the MDG syntax is available in [4].

A.1 The MDG Syntax

The `STRIP_DISJ_list` function is used to extract each disjunct and store it in a list:

```
⊢def   (STRIP_DISJ_list (DISJ a (CONJ1 b)) =
       (STRIP_CONJ_list a):: STRIP_DISJ_list (CONJ1 b) ) ∧
       (STRIP_DISJ_list (DISJ c d) = (STRIP_CONJ_list c) :: STRIP_DISJ_list (d)) ∧
       (STRIP_DISJ_list (CONJ1 b) = [(STRIP_CONJ_list b)] )
```

And `STRIP_CONJ_list` function is used to extract both side of equations and store them in the inner sublist as shown below:

```
⊢def  (STRIP_CONJ_list (CONJ a  (Eqn  b)) =
          (both_side_eq a):: (STRIP_CONJ_list (Eqn  b))) ∧
      (STRIP_CONJ_list (CONJ c  (d)) =
          (both_side_eq c) :: STRIP_CONJ_list (d)) ∧
      (STRIP_CONJ_list (Eqn b) = [ both_side_eq b])
⊢def  both_side_eq a = [ left_eq a; right_eq a]
```

The left_eq and right_eq functions extract the LHS and RHS of an equation. Similarly, STRIP_Fun function is used to extract the arguments of abstract and cross functions and store them in a list as defined below:

```
⊢def  (STRIP_Abst_Fun a = STRIP_Abst(FLAT(STRIP_DF_Abst a)))
⊢def  (STRIP_Cross_Fun a = STRIP_Cross(FLAT(STRIP_DF_Cross a)))
⊢def  (STRIP_Fun a = FLAT([(STRIP_Cross_Fun a) ; (STRIP_Abst_Fun a)]))
```

The STRIP_INV_DF function is used to map a list of lists to a DF format and is defined as:

```
⊢def  (STRIP_INV_DF (h1::t1) (h2::t2) (h3::t3) (h4::t4) =
          if ((h1::t1) = [[["TRUE"]]]) then (TRUE)
          else if((h1::t1) = [[["FALSE"]]]) then (FALSE)
          else
              (DF1 (build_DISJ (h1::t1) (h2::t2) (h3::t3) (h4::t4)))) ∧
      (STRIP_INV_DF (h1::t1) (h2::t2) [] (h4::t4) =
          if ((h1::t1) = [[["TRUE"]]]) then (TRUE)
          else if((h1::t1) = [[["FALSE"]]]) then (FALSE)
          else
              (DF1 (build_DISJ (h1::t1) (h2::t2) [] (h4::t4)))) ∧
      (STRIP_INV_DF (h1::t1) [] [[[]]] (h4::t4) =
          if ((h1::t1) = [[["TRUE"]]]) then (TRUE)
          else (FALSE))
```

where build_DISJ function is the inverse of the STRIP_DISJ_list function and used to build each disjunct from list as shown below:

```
⊢def  (build_DISJ (h1::t1) (h2::t2) (h3::t3) (h4::t4) =
          if (TL(h1::t1)=[]) then
              (CONJ1  (build_CONJ h1 h2 h3 h4))
          else
              ((DISJ (build_CONJ h1 h2 h3 h4)) ((build_DISJ t1 t2 t3 t4)))) ∧
      (build_DISJ (h1::t1) (h2::t2) [] (h4::t4) =
          if (TL(h1::t1)=[]) then
              (CONJ1  (build_CONJ h1 h2 [] h4))
          else
              ((DISJ (build_CONJ h1 h2 [] h4)) ((build_DISJ t1 t2 [] t4))))
```

Similarly, `build_CONJ` function is the inverse of the `STRIP_CONJ_list` function and used to build each conjunct from list format. Also, the `build_eq` function is used to build the equation from the list as defined below:

```
⊢def  (build_eq [ v1;v2] [ a1;a2] [] [ c1] =
          if (c1="EQUAL1") then
               (EQUAL1 (Concrete_Variable v1 a1) (Individual_Constant v2 a2))
          else if (c1="EQUAL4") then
               (EQUAL4 (Abstract_Variable  v1 a1) (Abstract_Variable  v2 a2))
          else
               (EQUAL5 (Abstract_Variable  v1 a1) (Generic_Constant v2 a2 ))) ∧
      (build_eq [ v1;v2] [ a1;a2] [ b1;b2] [ c1] =
          if(c1="EQUAL2") then
               (EQUAL2 (Abstract_Variable  v1 a1) (Abstract_Function v2 b2 a2))
          else
               (EQUAL3 (Cross_Function v1 b2 a1) b2 (Individual_Constant v2 a2)))
```

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)
```

A.2 The Conjunction Operation

In case of cross-function, SPLIT is used for substitution:

```
⊢def   (SPLIT _ [] = [] ) ∧
       (SPLIT [] _ = [] ) ∧
       (SPLIT (hd1::tl1) (hd2::tl2)=
            if (IS_EL (HD(TL hd1)) (HD_list (hd2::tl2))) then
                if ( HD(TL hd1) = (HD hd2) ) then
                    (HD hd1 :: (TL hd2)) :: SPLIT tl1 (hd2::tl2)
                else
                    SPLIT (hd1::tl1) (tl2)
            else
                hd1 :: SPLIT tl1 (hd2::tl2))
```

The position of RHS equation is specified by `position` function to check the order:

```
⊢def  (pos i [] x = 0) ∧
     (pos i (h::t) x = if h=x then (i+1) else pos (i+1) t x)
⊢def   position alist = pos 0 alist
```

The logical conjunction HOL function:

```
⊢def  ∀df1 df2. CONJ_LOGIC df1 df2 =
        (if df1 = TRUE then DISJ_LIST (STRIP_DF_list df2)
        else (if df2 = TRUE then DISJ_LIST (STRIP_DF_list df1)
        else (if df1 = FALSE then DISJ_LIST (STRIP_DF_list df1)
        else (if df2 = FALSE then DISJ_LIST (STRIP_DF_list df2)
        else
          DISJ_LIST (STRIP_DF_list df1) ∧ DISJ_LIST (STRIP_DF_list df2)))))
```

where the `DISJ_LIST` function is used to convert the list to DNF format. Then, the conjunction correctness is shown below in Theorem A.2.1.

Theorem A.2.1 *Conjunction Correctness*

`ASSUME:`

1. $df1$ and $df2$ are well-formed DF.

2. L is an order list equal to the union of $df1$ and $df2$ order lists.

Then, the correctness criteria for the proof of conjunction algorithm is shown in the following equation:

$$CONJ\_ALG(df1, df2, L) \equiv df1 \wedge df2.$$

and is proved in HOL as:

```
⊢  ∀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 SKETCH: By structural induction on $df1$ and $df2$ and rewriting rules. The goal is to prove the equivalence of MDG conjunction and HOL logical conjunction for these DF. This goal generates hundreds of subgoals since the proof takes all the cases of DF. The terminal cases when either df1 or df2 TRUE or FALSE are directly proved by applying the rewriting rule. Many base cases are generated, for example, the proof when both df1 and df2 are just an equation is shown by Lemma 1. □

Theorems regarding the terminal cases of conjunction algorithm:

```
lem_CONJ_TRUE_df2: ⊢  ∀df2. ∃L. Is_Well_Formed_DF TRUE ∧
  Is_Well_Formed_DF df2 ∧ (ORDER_LIST TRUE df2 = L) ⟹
  (CONJ_LOGIC TRUE df2 L = DISJ_LIST (CONJ_ALG TRUE df2 L))

lem_CONJ_FALSE_df2: ⊢  ∀df2. ∃L. Is_Well_Formed_DF FALSE ∧
  Is_Well_Formed_DF df2 ∧ (ORDER_LIST FALSE df2 = L) ⟹
  (CONJ_LOGIC FALSE df2 L = DISJ_LIST (CONJ_ALG FALSE df2 L))

lem_CONJ_DF1_TRUE: ⊢  ∀D. ∃L. Is_Well_Formed_DF (DF1 D) ∧
  Is_Well_Formed_DF TRUE ∧ (ORDER_LIST (DF1 D) TRUE = L) ⟹
  (CONJ_LOGIC (DF1 D) TRUE L = DISJ_LIST (CONJ_ALG (DF1 D) TRUE L))

lem_CONJ_DF1_FALSE: ⊢  ∀D. ∃L. Is_Well_Formed_DF (DF1 D) ∧
  Is_Well_Formed_DF FALSE ∧ (ORDER_LIST (DF1 D) FALSE = L) ⟹
  (CONJ_LOGIC (DF1 D) FALSE L=DISJ_LIST (CONJ_ALG (DF1 D) FALSE L))
```

Another base case representing the conjunction of two equations:

```
Lemma 1: CONJ1 (Eqn a) CONJ1 (Eqn b) ⊢  ∀a b. ∃L.
  Is_Well_Formed_DF (DF1 (CONJ1 (Eqn a))) ∧
  Is_Well_Formed_DF (DF1 (CONJ1 (Eqn b))) ∧
  (ORDER_LIST (DF1 (CONJ1 (Eqn a))) (DF1 (CONJ1 (Eqn b))) = L) ⟹
   CONJ_LOGIC (DF1 (CONJ1 (Eqn a))) (DF1 (CONJ1 (Eqn b))) L =
    DISJ_LIST CONJ_ALG (DF1 (CONJ1 (Eqn a))) (DF1 (CONJ1 (Eqn b))) L)
```

A.3 The RelP Operation

The logical RelP function is embedded as shown below:

```
⊢def  ∀df1 df2 L. (RelP_LOGIC df1 df2 L =
                   EXISTS_LIST L (CONJ_LOGIC df1 df2) )
```

where EXISTS_LIST function is similar to EXIST_QUANT. Then, the RelP correctness is:

Theorem A.3.1 *Relational Product Correctness*

ASSUME:

1. $df1$ and $df2$ are well-formed DF.

2. $L1$ is an order list equal to the union of $df1$ and $df2$ order lists.

3. $L2$ is the primary variables of both $df1$ and $df2$.

Then, the correctness criteria for the proof of RelP algorithm is shown in the following equation:

if $R = (EXISTS\_QUANT\ (df1 \wedge df2)\ L2)$ *then :*

$$RelP\_ALG(df1, df2, L) \equiv R.$$

and is proved in HOL as:

```
⊢  ∀df1 df2. ∃L1. ∃L2.
 (Is_Well_Formed_DF df1) ∧ (Is_Well_Formed_DF df2) ∧
 (ORDER_LIST df1 df2=L1) ∧ (IS_PRIMARY_VAR_LIST L2 df1 df2) ⟹
 ((RelP_LOGIC df1df2 L1 L2) = DISJ_LIST(RelP_ALG df1 df2 L1 L2))
```

PROOF SKETCH: By structural induction on $df1$ and $df2$ and rewriting rules. The MDG RelP of $df1$ and $df2$, and the HOL logical $(EXISTS\_QUANT(df1 \wedge df2)L2)$, are equivalent. □

A.4 The Disjunction Operation

The logical definition for the disjunction algorithm is specified by DISJ_LOGIC function:

```
⊢def  ∀df1 df2 L. DISJ_LOGIC df1 df2 L =
  (if df1 = TRUE then DISJ_LIST (STRIP_DF_list df1)
  else (if df2 = TRUE then DISJ_LIST (STRIP_DF_list df2)
  else (if df1 = FALSE then DISJ_LIST (STRIP_DF_list df2)
  else (if df2 = FALSE then DISJ_LIST (STRIP_DF_list df1)
  else (if FLAT (STRIP_ABS_DF df1) =
           FLAT (STRIP_ABS_DF df2) then
            DISJ_LIST (STRIP_DF_list df1) ∨ DISJ_LIST (STRIP_DF_list df2)
  else F)))))
```

The disjunction correctness is:

Theorem A.4.1 *Disjunction Correctness*

ASSUME:

1. $df1$ and $df2$ are well-formed DF.

2. L is an order list equal to the union of $df1$ and $df2$ order lists.

Then, the correctness criteria for the proof of disjunction algorithm is shown in the following equation:

$$DISJ\_ALG(df1, df2, L) \equiv df1 \vee df2.$$

and is proved in HOL as:

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

PROOF SKETCH: By structural induction on $df1$ and $df2$ and rewriting rules. The MDG disjunction of $df1$ and $df2$, and the HOL logical disjunction of $df1$ and $df2$, are equivalent. □

A.5 The PbyS Operation

The DF_PbyS2 function checks the existence of the first equation of df1 in df2. If it exists then the function will discard it (subsumed by df2). Otherwise the equation is added to the result (cannot be subsumed).

```
⊢def  (PbyS_2 [] (hd2::tl2) _ _ = [] ) ∧
      (PbyS_2 (hd1::tl1) [] _ _ =  (hd1::tl1)) ∧
      (PbyS_2 (hd1::tl1) (hd2::tl2) L4 (hd5::tl5) =
            if ((PbyS_3 (hd1::tl1) hd2 L4 hd5) = [])  then
                 []
            else
                 PbyS_2 (hd1::tl1) tl2 L4 tl5
```

DF_PbyS3 function is similar to DF_PbyS2 function and defined as:

```
⊢def  (PbyS_3 [] (hd2::tl2) _ _ = [] ) ∧
      (PbyS_3 (hd1::tl1) [] _ _ =  (hd1::tl1)) ∧
      (PbyS_3 (hd1::tl1) (hd2::tl2) L4 (hd5::tl5) =
            if (IS_EL hd1 (hd2::tl2))  then
                 PbyS_3 tl1 (hd2::tl2) L4 (hd5::tl5)
            else if (IS_EL (HD hd1) (HD_list (hd2::tl2))) then
                 if (IS_EL (HD hd1) (hd5::tl5)) then
                     PbyS_3 tl1 (hd2::tl2) L4 (hd5::tl5)
                 else
                     (hd1::tl1)
            else
                 (hd1::tl1))
```

The logical pruning by subsumption function is embedded as shown below:

```
⊢def  ∀df1 df2. PbyS_LOGIC df1 df2 =
        if (df1 = TRUE) then [[["FALSE"]]]
        else if (df2 = TRUE) then [[["FALSE"]]]
        else if (df1 = FALSE) then DISJ_LIST (STRIP_DF_list df1)
        else if (df2 = FALSE) then (STRIP_DF_list df1)
        else if (IS_EL L1 L2) then
            F
        else
            DISJ_LIST (STRIP_DF_list df1) ∧
            DISJ_LIST (STRIP_DF_list df2)
```

To prove the correctness of PbyS operation, we need to verify Theorem A.5.1.

Theorem A.5.1 *Pruning by Subsumption Correctness*

ASSUME:

*1. df*1 *and df*2 *are well-formed DF.*

2. L is an order list equal to the union of df1 and df2 order lists.

Then, the correctness criteria for the proof of PbyS algorithm is shown in the following:

if $P1 = PbyS\_ALG(df1, df2, L)$ *and* $P2 = (EXIST\_QUANT\ df2\ L2)$

then : $DISJ\_ALG(P1, P2, L) \equiv df1 \vee P2$.

and is proved in HOL as:

```
⊢ ∀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 SKETCH: By structural induction on $df1$ and $df2$ and rewriting rules. The MDG disjunction of $PbyS\_ALG(df1, df2, L)$ and $(EXIST\_QUANT df2 L2)$, is equivalent to the HOL disjunction of $df1$ and $(EXISTS\_QUANT df2 L2)$. □

The PbyS operation is used to check whether a set of states is a subset of another set of states as shown in Lemma 2. Let df1, df2 be two DFs of type $U \rightarrow V$, then we say that df1 and df2 are *equivalent* DFs if `PbyS(df1,df2,L) = PbyS(df2,df1,L)`.

```
Lemma 2: 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)
```

A.6 The Reachability Analysis

The MDG reachability analysis `Re_An` is defined in HOL by calling `RA_n` with the circuit parameters:

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

where the variables (v=I I_F Q_F Tr Tr_F Tr_A In Ren L R R_F R_A) are extracted from the initialization step (algorithm inputs).

The predicate 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 ))
```

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.

The Reach_Step computes the next reachable state by applying successively Union_Step which calls Next_State and Frontier_Step:

```
⊢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)
```

The Next_State computes the set of next states by using the RelP operator:

```
⊢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 renaming substitution and the set of inputs and state variables over the resulting DF are quantified (the Next_State function).

The Frontier_Step is used to check if all the states reachable by the machine are already visited. The implementation uses the PbyS 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)
```

Finally, Union_Step 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))
```

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

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

Biography

- **Education**
  - **Concordia University:** Montreal, Quebec, Canada

    Ph.D candidate, in Electrical and Computer Engineering, 09/03-present.
  - **Jordan University of Science and Technology (J.U.S.T.)**, Jordan:

    M.Sc., in Electrical and Computer Engineering, 09/94 - 10/96.
  - **Jordan University of Science and Technology (J.U.S.T.)**, Jordan:

    B. Eng., in Electrical and Computer Engineering, 09/89 - 02/94.
- **Work Experience**
  - **Research Assistant:** 02/05-present

    Hardware Verification Group (HVG), Concordia University
  - **Teaching Assistant:** 09/04-present

    Department of Electrical and Computer Engineering, Concordia University, Montreal, Canada.
  - **Lecturer:** 11/97-09/03

    Computer Science Department, King Faisal University, Saudi Arabia.
  - **Lecturer:** 09/96-09/97

    Computer Engineering Department, Hijjawi Faculty, Yarmouk University, Jordan.
  - **Teaching Assistant:** 09/94-09/96

    Department of Electrical Engineering, Jordan University of Science and Technology (J.U.S.T.), Jordan.

– **Computer Engineer:** 02/94-09/94

Spectra for Engineering & Computer Company (SEC), Jordan.

• Publications

Journal Papers

1. Saed Abed, Otmane Ait Mohamed and Ghiath Al Sammane, "Automatic Verification of Reduction Techniques in High Order Logic", Submitted to *IEEE Transactions on Computers.*

2. Saed Abed, Otmane Ait Mohamed and Ghiath Al Sammane, "Towards a Reachability Approach by Combining HOL Induction and Multiway Decision Graphs", Submitted to *Journal of Computer Science and Technology (JCST).*

3. Saed Abed, Otmane Ait Mohamed and Ghiath Al Sammane, "On The Integration of Decision Diagrams in High Order Logic Based Theorem Provers: a Survey", *Journal of Computer Science*, Science Publication, Vol. 3, No. 10, Dec. 2007, pp. 810-817.

4. M. S. Ibbini and Sa'ed R. Alawneh, "Closed-Loop Control System Robustness Improvement by Parameterized State Feedback", *IEE Proc.- Control Theory Appl.*, Vol. 145, No. 1, January 1998, pp. 33-40.

Conference Papers

1. Saed Abed, Otmane Ait Mohamed and Ghiath Al Sammane, "The Performance of Combining Multiway Decision Graphs and HOL Theorem Prover", To appear *In Proc. Languages for Formal Specification and Verification, Forum on Specification & Design Languages (IEEE FDL'08),*

Stuttgart, Germany, September 23-25, 2008.

2. Y. Mokhtari, Saed Abed, Otmane Ait Mohamed, S. Tahar and X. Song, "A New Approach for the Construction of Multiway Decision Graphs", To appear *In Proc. 5th International Colloquium on Theoretical Aspects of Computing (ICTAC'08)*, Lecture Notes in Computer Science, Istanbul, Turkey, September 1-3 ,2008.

3. Saed Abed and Otmane Ait Mohamed, "LCF-style Platform based on Multiway Decision Graphs", *In Proc. 17th Workshop on Functional and (Constraint) Logic Programming (WFLP'08)*, Siena, Italy, July 3-4, 2008, pp. 139-153.

4. Saed Abed, Otmane Ait Mohamed and Ghiath Al Sammane, "Multiway Decision Graphs Reduction Approach based on the HOL Theorem Prover", *In Proc. Second International Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS'08)*, The British Computer Society, Leeds, U.K, July 2-3, 2008, pp. 1-10.

5. Saed Abed and Otmane Ait Mohamed, "The MDG-HOL Platform for Automatic Verification", *In Proc. 10th Maghrebian Conference on Software Engineering and Artificial Intelligence (MCSEAI'08)*, Algeria, April, 2008, pp. 659-664.

6. Saed Abed, Otmane Ait Mohamed and Ghiath Al Sammane, "Reachability Analysis using Multiway Decision Graphs in the HOL Theorem Prover", *In Proc. ACM SAC '08*, Fortaleza, Brazil, March 2008, pp. 333-338.

7. Saed Abed, Otmane Ait Mohamed and Ghiath Al Sammane, "Integrating SAT with Multiway Decision Graphs for Efficient Model Checking", *In Proc. IEEE International Conference on Microelectronics (ICM)*, Egypt,

Dec. 2007.

8. Saed Abed, Otmane Ait Mohamed and Ghiath Al Sammane, "A High Level Reachability Analysis using Multiway Decision Graph in the HOL Theorem Prover", *International Conference on Theorem Proving in Higher Order Logics (TPHOLs'07): B-Track Proceedings*, Kaiserslautern, Germany, Sep. 2007, pp. 1-17.

9. Ghiath Al Sammane, Saed Abed and Otmane Ait Mohamed, "High Level Reduction Technique for Multiway Decision Graphs Based Model Checking", *In Proc. First International Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS 2007)*, Algeria, May 2007, pp. 1-14.

10. Donglin Li, Otmane Ait-Mohamed and Sa'ed Abed, "Towards First-Order Symbolic Trajectory Evaluation", *The 37th International Symposium on Multiple-Valued Logic (ISMVL 2007)*, May 2007, Oslo, Norway, pp. 1-7.

11. Saed Abed and Otmane Ait Mohamed, "Embedding of MDG Directed Formulae in HOL Theorem Prover", *In Proc. 9th Maghrebian Conference on Software Engineering and Artificial Intelligence (MCSEAI'06)*, Morocco, Dec., 2006, pp. 659-664.

Technical Reports

1. Y. Mokhtari, S. Abed, O. Ait Mohamed, S. Tahar and X. Song, A New Approach for the Construction of Multiway Decision Graphs; Technical Report 2008-3-Abed, Concordia University, Department of Electrical and Computer Engineering, June 2008.

2. Saed Abed and Otmane Ait Mohamed, Formalizing MDGs Basic Operations as Inference Rules in the HOL Theorem Prover; Technical Report 2008-2-Abed, Concordia University, Department of Electrical and Computer Engineering, Jan. 2008.

3. Saed Abed, Otmane Ait Mohamed and Ghiath Al Sammane, HOL Based Reduction Techniques for MDGs Model Checking; Technical Report 2008-1-Abed, Concordia University, Department of Electrical and Computer Engineering, Jan. 2008.

4. Saed Abed, Otmane Ait Mohamed and Ghiath Al Sammane, On the Embedding and Verification of Multiway Decision Graph in HOL Theorem Prove; Technical Report 2007-1-Abed, Concordia University, Department of Electrical and Computer Engineering, Feb. 2007.

Honors & Awards

1. Dean Teaching Scholarship, ECE Dept., Concordia University: 2006-2007.

2. Teaching Scholarship, ECE Dept., Jordan University of Science and Technology to get an M.Sc. in Computer Engineering: 1994 - 1996.

3. Ministry of Higher Education Scholarship to get a B.Sc. in Computer Engineering at Jordan University of Science and Technology: 1989 - 1994.