Providing a Formal Linkage between MDG and HOL Based on a Verified MDG System

A thesis submitted to Middlesex University
in partial fulfilment of the requirement for the degree of
Doctor of Philosophy

Haiyan Xiong

School of Computing Science

Middlesex University

January 2002

Abstract

Formal verification techniques can be classified into two categories: deductive theorem proving and symbolic state enumeration. Each method has complementary advantages and disadvantages. In general, theorem provers are high reliability systems. They can be applied to the expressive formalisms that are capable of modeling complex designs such as processors. However, theorem provers use a glass-box approach. To complete a verification, it is necessary to understand the internal structure in detail. The learning curve is very steep and modeling and verifying a system is very time-consuming. In contrast, symbolic state enumeration tools use a black-box approach. When verifying a design, the user does not need to understand its internal structure. Their advantages are their speed and ease of use. But they can only be used to prove relatively simple designs and the system security is much lower than the theorem proving system. Many hybrid tools have been developed to reap the benefits of both theorem proving systems and symbolic state enumeration systems. Normally, the verification results from one system are translated to another system. In other words, there is a linkage between the two systems. However, how can we ensure that this linkage can be trusted? How can we ensure the verification system itself is correct?

The contribution of this thesis is that we have produced a methodology which can provide a formal linkage between a symbolic state enumeration system and a theorem proving system based on a verified symbolic state enumeration system. The methodology has been partly realized in two simplified versions of the MDG system

(a symbolic state enumeration system) and the HOL system (a theorem proving system) which involves the following three steps. First, we have verified aspects of correctness of two simplified versions of the MDG system. We have made certain that the semantics of a program is preserved in those of its translated form. Secondly, we have provided a formal linkage between the MDG system and the HOL system based on importing theorems. The MDG verification results can be formally imported into HOL to form the HOL theorems. Thirdly, we have combined the translator correctness theorems with the importing theorems. This combination allows the low level MDG verification results to be imported into HOL in terms of the semantics of a high level language (MDG-HDL). We have also summarized a general method which is used to prove the `existential theorem` for the specification and implementation of the design. The feasibility of this approach has been demonstrated in a case study: the verification of the correctness and usability theorems of a vending machine.

Acknowledgments

I have been very fortunate to have had Dr. Paul Curzon, Prof. Ann Blandford and Prof. Sofiène Tahar as my supervisors. I am deeply grateful for their support and encouragement throughout my Ph.D studies. I am most indebted to them for the considerable amount of time they each devoted to me in my research work. I extend my deepest thanks especially to Dr. Paul Curzon, without whose invaluable guidance I could not have completed this work.

In this thesis, several of the chapters are based on publications that were produced in the course of this research. The papers published jointly with my supervisors were all first-authored by me, and all report on my own work, completed under their supervision [78 - 83]. The work reported in Chapter 5 realized a general idea of Curzon and Tahar [80]: I formalized that idea in HOL. The work reported in Chapter 8 takes an example that was originally developed by Curzon and Blandford [24], and applies the approach developed within this thesis to that same example. The MDG verification was completed with the help of Tahar. All the HOL proof is my own work, again completed under their supervision.

I would like to thank people in the Automated Reasoning Group in Cambridge and the MDG group in Montreal, Prof. Mike Gordon, Dr. Konrad Slind, Dr. Michael Norrish, Joe Hurd, Dr. Richard Boulton and Prof. Tom Melham. When I have needed help they have always lent me a hand. I have benefitted so very much from their vast knowledge and insight.

I am particularly thankful to Dr. Wai Wong, who not only introduced me to this field, but also provided a great deal of assistance.

Many thanks to Sardia, who provided fabulous administrative support, and to Leonard, who was always available whenever I had problems with my computer.

I would like to reserve my deepest thanks for my parents for their perpetual love and encouragement, and to my husband and my son for their sacrifices and patience. I can never thank them enough.

Lastly, I would like to acknowledge the support obtained from the School of Computing Science, Middlesex University and EPSRC grant GR/M45221.

Haiyan Xiong

Contents

List of Figures

Chapter 1

Introduction

Formal methods are the application of applied mathematics - formal logic - to the design and analysis of computer systems. Generally, formal verification techniques can be classified into two categories: deductive theorem proving and symbolic state enumeration. In deductive theorem proving systems, the correctness condition for a design is represented as a theorem in a mathematical logic, and a mechanically checked proof of this theorem is generated using a general-purpose theorem prover. In symbolic state enumeration systems, the design being verified is represented as a decision diagram. Techniques such as reachability analysis are used to automatically verify given properties of the design or machine equivalence. Much of this work is based on Binary Decision Diagrams (BDD) [4] [11].

Deductive theorem proving systems use interactive proof methods. In these systems, an implementation and its behavioral specification are represented as first-order or higher-order logic formulas. The user interactively constructs a formal proof which proves a theorem stating the correctness of this implementation. Theorem proving systems are naturally deductive process systems. They allow a hierarchical verification method to be used to model the overall functionality of designs with complex datapaths. They are very general in their applications. The theorems can not only be used to formalize a specific design but also can be abstracted as a general situation of this class of design. Theorem proving systems are semi-automated. To complete a verification, experts with good knowledge of the internal structure of the design are required to guide the proof searching process. This enables the designer to gain greater insight into the system and thus achieve better designs. However, the learning curve is very steep, modelling and verifying a system is very time-consuming. This is the major difficulty to applying the theorem proving systems in industry.

In contrast, symbolic state enumeration systems are automated decision diagram approaches. In this kind of approach, an implementation and its behavioral specification are represented as decision diagrams. A set of algorithms is used to efficiently manipulate the decision diagrams so as to get the correctness results. The introduction to the BDD based method by Hu [47] may be taken as a good reference. In contrast to the theorem prover, symbolic state enumeration verification is a relatively modest activity. It normally deals with a single model rather than the whole design. The symbolic state enumeration verification approach can be viewed as a black-box approach. During the verification, the user does not need to understand the internal structure of the design. The strength of this approach is its speed and ease of use. However, it does not scale well to complex designs since it uses non-hierarchy state-based descriptions of the design. An increase in the number of design components can result in the state space growing exponentially.

In the 1990s, the efficiency breakthrough in symbolic state enumeration was such that industry has successfully applied symbolic state enumeration tools in digital circuit synthesis and verification. Since then, more and more tools have been developed including Spin [45], MDG [20], STE [72] and so on. Although they have been very successfully used in industry, there are still many deficiencies in the currently available symbolic state enumeration tools. Although the symbolic state enumeration based tools can be applicable to circuits of considerable size, they still do not scale up sufficiently. However, the theorem proving systems can be applied to large designs in theory, although in practice it is time consuming. One solution is to combine these two kind of systems to reap the advantages of both. This combination allows the fully automated proof tools to rely on a theorem proving system and the increasing size and complexity of a design can be handled in practice.

Recently, there has been a great deal of work concerned with combining the theorem proving and symbolic state enumeration systems. A common approach to combining proof tools is to use an symbolic state enumeration system as an oracle to provide results to the theorem proving system. The issue in such work is to guarantee that the results provided by external tools are theorems within the theory of the proof system. In other words, an oracle is used to receive problems and return answers. For example, the HOL system provides approaches for tagging theorems that are dependent on the correctness of external verification tools. An oracle can be built in the HOL system is viewed as a plug-in. This brings about two questions.

1. Can we ensure the automated verification system produces the correct results?
2. Have the verification results from an automated verification system been correctly converted into a valid theorem in the current theory of the theorem proving system?

The research describe here investigates the answers to the above two questions. In fact, some symbolic state enumeration based systems such as MDG [20] consist of a series of translators and a set of algorithms. Higher level languages such as hardware description languages are used to describe the specification and implementation of the design. The specification and implementation are then translated into the decision diagrams via intermediate languages. The algorithms in the system are used to efficiently and automatically deal with the decision diagrams so as to obtain the correctness results. We need to verify the translators and algorithms in order to get the answer of the first question. For solving the second question, we need to formally justify the correctness results, which are obtained from a symbolic state enumeration system, into a theorem prover, to ensure the correctness of the theorem creation process.

In this thesis, we will produce a methodology, which can provide a formal linkage between a theorem proving system and a symbolic state enumeration system based on a verified symbolic state enumeration system, to ensure the correctness of the theorem creation process. We first need to verify aspects of correctness of the symbolic state enumeration system in an interactive theorem proving system. We then need to prove the translators and algorithms to ensure the correctness of the system. By combining the translator correctness theorems with the importing theorems, the verification results from the state enumeration system can be imported into the theorem proving system in terms of the semantics of high level language (HDL) rather than low level language (decision graph). We also need to summarize a general method to prove the `existential theorem` of the design, which is needed for importing sequential verification result into the theorem proving system.

We will partly realize the methodology in the HOL system and two simplified versions of the MDG system. We will prove the correctness of aspects of the simplified versions of the MDG system and provide a formal linkage between the HOL system and the simplified versions of the MDG system. Lessons from the research could be applicable to other related systems. We chose HOL and MDG because this research is part of a large project in collaboration with the Hardware Verification group at Concordia University. They are developing a hybrid system (MDG-HOL) [54] [53] [66] which combines the MDG system and the HOL system. Our aim is different to theirs. We are not developing a practical tool. We are doing theoretical research about how to verify the MDG system and provide a formal linkage between the HOL system and the MDG system. Our deep embedding semantics is in terms of the specification of the MDG system. Since we will consider the simplified versions of the MDG system, in the rest of this thesis, we will refer to the simplified versions of the MDG system as 'the MDG system' except in the section 1.1.

In the research, we first consider verifying the translation phases of the MDG system using the HOL system and obtain a series of correctness theorems. By combining those theorems, we obtain that the semantics of a low level MDG program equals the semantics of a high level MDG-HDL program (the MDG input language). We then consider how to formally import the MDG verification results to a form that can be used in the HOL system. We formalize the MDG verification results in terms of the semantics of the low level MDG program and turn them into HOL to form the HOL theorems. By combining the translation correctness theorems with the importing theorems, we obtain theorems which convert the low level MDG verification results into HOL to form the HOL theorem based on the semantics of the MDG input language. In other words, this combination allows the imported theorem to be in terms of the semantics of the MDG-HDL. For easily importing the MDG results into HOL for sequential verification, we summarize a general way to prove the `existential theorem` (a theorem which has form: $\forall$ `ip.` $\exists$ `op. C ip op`). All the theorems in this thesis written with $\vdash_{thm}$ have been proved in HOL.

The structure of the rest of this chapter is as follows: In sections 1.1 and 1.2, we will briefly introduce the MDG system and the HOL system respectively. An overview of the research will be given in section 1.3. Finally, an outline of this thesis will be presented in the last section.

1.1 The MDG System

The full MDG system is an automated verification tool for hardware verification. It uses a new class of decision graphs called Multiway Decision Graphs, which subsume the class of Bryant's Reduced and Ordered Binary Decision Diagrams (ROBDD) [12] while accommodating abstract sorts and uninterpreted function symbols.

A multiway decision graph (MDG) is a finite directed acyclic graph G where the leaf nodes are labeled by formulas, the internal nodes are labeled by terms and the edges issuing from an internal node, N, are labeled by terms of the same sort as the label of N. Such a graph represents a formula defined inductively as follows:

1. If G consists of a single leaf node labeled by a formula P, then G represents P,

2. If G has a root node labeled A with edges labeled $B_1 ... B_n$ leading to subgraphs $G_1{}' ... G_n{}'$, and if each $G_i{}'$ represents a formula P_i, then G represents the formula $\vee_{1<\mathrm{i}<\mathrm{n}}((A = B_i) \wedge P_i)$.

In fact, when an MDG has been constructed as a graph, it must obey the restrictions that any path from the root to leaf yields a canonical representation. Like ROBDDs, an MDG must be reduced and ordered. Unlike ROBDDs, all the variables used in an MDG must have appropriate sort, and sort definitions must be provided for all functions. MDG can also 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 underlying logic of MDG is a subset of many-sorted first-order logic with a distinction between concrete and abstract sorts. A concrete sort has an enumeration while an abstract sort does not. Therefore, a data signal can be represented by a single variable of abstract sort and a data operation can be represented by an uninterpreted function symbol. It partially fulfills the aim of interactive verification to verify hardware designs automatically at a high level of abstraction. It also lifts many ROBDD techniques from the boolean domain to a more abstract domain. In particular, a data signal in an MDG is represented by a single variable of abstract sort rather than a vector of boolean variables, and the data represents an operation by an uninterpreted function symbol. Therefore, MDGs are more compact than ROBDDs for circuits having a datapath, and this greatly increases the range of circuit that can be proved.

The MDG package has been implemented in Prolog. Algorithms such as disjunction, relational product (combination of conjunction and existential quantification), pruning-by-subsumption (for testing of set inclusion) and reachability analysis (using abstract implicit enumeration) have been developed. Applications for hardware verification such as combinational verification, sequential verification, invariant checking and model checking are provided.

1.2 The HOL System

The HOL system is an LCF (Logic of Computable Functions) style proof system. It uses higher-order logic to model and verify a system. There are two main different proof methods: forward and backward proof. 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. In backward proof, the user sets the desired theorem as a goal. Small programs written in SML [65] called tactics and tacticals are applied to breaking the goal into a list of subgoals. Tactics and tacticals are repeatedly applied to the subgoals until they can be resolved. A justification function is also created mapping a list of theorems corresponding to subgoals to a theorem that solves the goal. In practice, forward proof is often used within backward proof to convert each goal's assumptions to 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. More complex inference rules and tactics must ultimately call a series of primitive rules to do the work. In this way, the ML type system protects the HOL logic from the 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.

HOL 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, simpLib, decisionLib and bossLib`. These libraries integrate rewriting, conversion and decision procedures that automate a proof. They free the user from performing low-level proof.

1.3 Overview of the Research

The intention of our research is to explore a way of increasing the degree of trust of the MDG system and provide a formal linkage between the HOL system and the MDG system as shown in Figure 1.1. This work can be divided into three steps. (a) We must verify the correctness of the MDG system using the HOL system. It consists of two phases–(1) verification of the translators [82] and (2) verification of the algorithms. (b) We then must prove theorems (step 3 in Figure 1.1), which formally convert the verification results of different MDG applications into the traditional HOL hardware verification theorems [80]. (c) By combining the correctness theorems (theorems obtain from step 1, 2 in Figure 1.1) of the verification of the MDG system with the importing theorems (theorems obtain from step 3 in Figure 1.1), the MDG verification results can be imported into HOL in terms of the MDG input language.

During this study, we concentrate on the verification of the translation phase of the MDG system (step 1, Figure 1.1) using the HOL theorem prover and importing the MDG results into HOL to form the HOL theorems (step 3, Figure 1.1) [80]. Step 2 is similar to Chou and Peled's work [17] which verifies a partial-order reduction technique for model checking. Verifying the algorithms is beyond the scope of this thesis, as we are primarily concerned with the linkage and how it could be combined with the correctness theorems and importing theorems. We outline the methodology of the whole story and emphasize the importation process of the hybrid system. We not only verify the correctness of aspects of the MDG system in HOL, but also formally import the MDG results into HOL to form the HOL theorems based on the

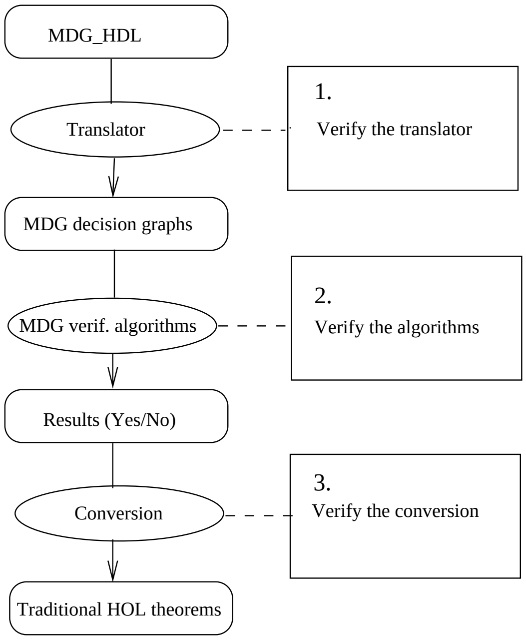


Figure 1.1: Overview of the Research

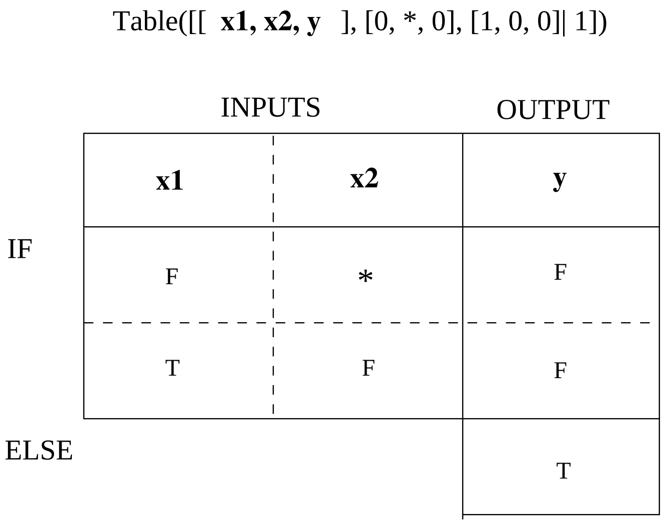
semantics of the high level MDG input language (MDG-HDL) [86] rather than the semantics of the low level language. Since we use a deep embedding semantics, the translator correctness theorems can be combined with other translator correctness theorems and the importing theorems. These combinations allow the low level MDG results to be converted into a form that can be easily reasoned about in HOL based on the semantics of MDG-HDL. We also summarize the general method about proving the `existential theorem` to remove the burden from the user of the combined system. This theorem is needed for importing sequential verification result into the theorem proving system.

In the remainder of this section, we will briefly introduce the individual steps that we have undertaken: verifying the translator correctness theorems, proving the general importing theorems, combining the translator correctness theorems with the importing theorems on the basis of deep embedding semantics and proving the `existential theorem`. These will each be considered in detail in subsequent chapters.

1.3.1 Verifying the MDG Translators

The input language of the MDG system is a Prolog-style hardware description language (MDG-HDL) [86], which allows the use of abstract variables for representing data signals. It 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. In MDG, a circuit description file declares signals and their sort assignment, components network, outputs, initial values for sequential verification and the mapping between state variables and next state variables. In the components network, there is a large set of predefined components such as logic gates, flip-flops, registers and constants, etc. Among the predefined components there is a special component called a Table, which is used to describe a functional block in the implementation and specification. The Table constructor is similar to a truth table, but allows first-order terms in rows. It also allows the high-level description to construct ITE (If-Then-Else) formulas and CASE formulas. A table is essentially a series of lists, together with a single final default value. The first list contains variables and cross-terms. The last element of the list is the output of the table which must be a variable (either concrete or abstract). For example, a two input `AND` gate can be described as the `table` as shown in Figure 1.2. In the figure, "*" means "`don't care`". It states that if `x1` is equal to `false` and `x2` is `DON'T CARE` then the output `y` is equal to `false`, if `x1` is equal to `true` and `x2` is equal to `false` then the output `y` is equal to `false`, otherwise the output `y` is equal to `true`.

Most of the components in the MDG-HDL library are compiled into their own core MDG-HDL code (tabular codes) first. The core MDG-HDL program can then be compiled into an internal MDG decision graphs (MDGs). Some components, such as registers, are implemented directly in terms of MDGs. However, in theory these components also could be implemented as tables to provide general specification mechanism. We assume the MDG-HDL program is firstly translated into a core MDG-HDL program and then the core MDG-HDL program is translated into MDG.



(a) AND gate table in MDG−HDL and core MDG−HDL

Figure 1.2: The AND Table

Figure 1.3: Overview of the MDG Translation Phases

In this situation, the MDG system could be specified as in Figure 1.3.

Because the Table constructor allows the high-level description to construct ITE formulas and CASE formulas, the possible input value of the else condition is not listed in the `table` of the core MDG-HDL. For example, the possible input value for the else condition of the `AND` gate table should be that if `x1` is equal to `true` and `x2` is equal to `true` then the output `y` is equal to `true`. It is not contained in the table. However, an internal MDG decision graph is determined in terms of all possible input value of its table which could be represented as a formula representation. Therefore, the MDG system translates the core MDG-HDL program into its formula representation first. In the MDG formula representation program, the table not only contains the input value of the if condition, it also contains the possible input value of the else condition. For example, an `AND` gate can be described as shown in the Figure 1.4.

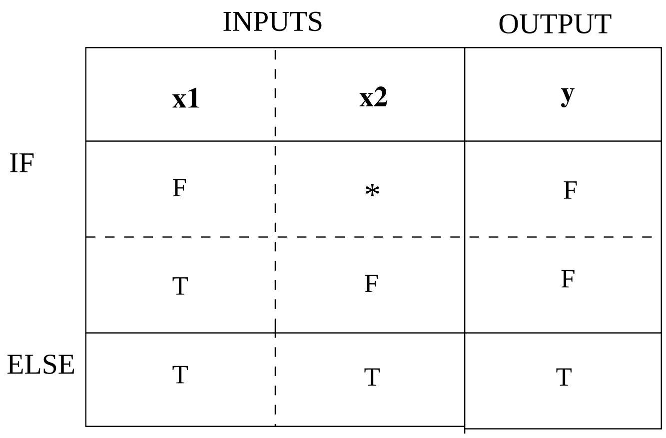


Figure 1.4: The `AND` Gate in the MDG Formula Representation

MDH−HDL $\xrightarrow{(1)}$ core MDG−HDL $\xrightarrow{(2)}$ MDG formula representation $\xrightarrow{(3)}$ MDG decision graphs

Figure 1.5: The MDG Translation Phases

In other words, the step (2) in Figure 1.3 could be further divided into two steps. The core MDG-HDL program is translated into the MDG formula representation first and the MDG formula representation program can then be translated into an internal MDG decision graph. Now, the MDG system could be specified as in Figure 1.5.

Adopting this approach makes the translation phase more amenable to verification. We are not verifying the actual MDG implementation. Rather our formalization of the translator is a specification of it. Once combined with a translator from core MDG-HDL to MDGs, it would be specifying the output required from the implementation. This would be used as the basis for verifying such an implementation. Effectively we split the problem of verifying the translator into the two problems of verifying that the implementation meets a functional specification, and that the functional specification then meets the requirement of preserving semantics. We are concerned with the latter step here. This split between implementation correctness and specification correctness was advocated by Chirica and Martin [16] with respect to compiler correctness.

In our research, we intend to verify the translation phase of the MDG system (Figure 1.5) based on the semantics of the MDG input language using the HOL theorem prover. As we mentioned above, the MDG system can be considered as a series of translators, translating between different intermediate languages, as shown in Figure 1.6. The verification process includes the following steps. Firstly, the syntax and the semantics of the subset MDG-HDL, core MDG-HDL, MDG formula representation and MDG decision graph will be defined. A set of functions, which translate the program from MDG-HDL to core MDG-HDL, from core MDG-HDL to the MDG formula representation and from the MDG formula representation to the MDG decision graph, will then be defined. For each program in MDG-HDL, core MDG-HDL or the MDG formula representation, the compilation operators are defined as functions, which return their core MDG-HDL, the MDG formula representation or MDG decision graph code. Translation functions `TransProgMC`, `TransProgCF` or `TransProgFM` are applied to each MDG-HDL program, core MDG-HDL program or the MDG formula representation so that the corresponding core MDG-HDL program, MDG formula representation program or MDG decision graph program is established. In other words, the relations of the translations can be represented as below:

```
   ∀  p.  TransProgMC p = the core MDG-HDL program
or
   ∀  p.  TransProgCF (TransProgMC p) =
                            the MDG formula representation program
or
   ∀  p.  TransProgFM (TransProgCF (TransProgMC p)) =
                            the MDG decision graph program
```

The standard approach to prove a translator between two languages is in terms of the semantics of the languages, shown in Figure 1.6. Essentially the translation should preserve the semantics of the source language. This has the traditional form of compiler specification correctness used in the verification of a compiler [16]. The

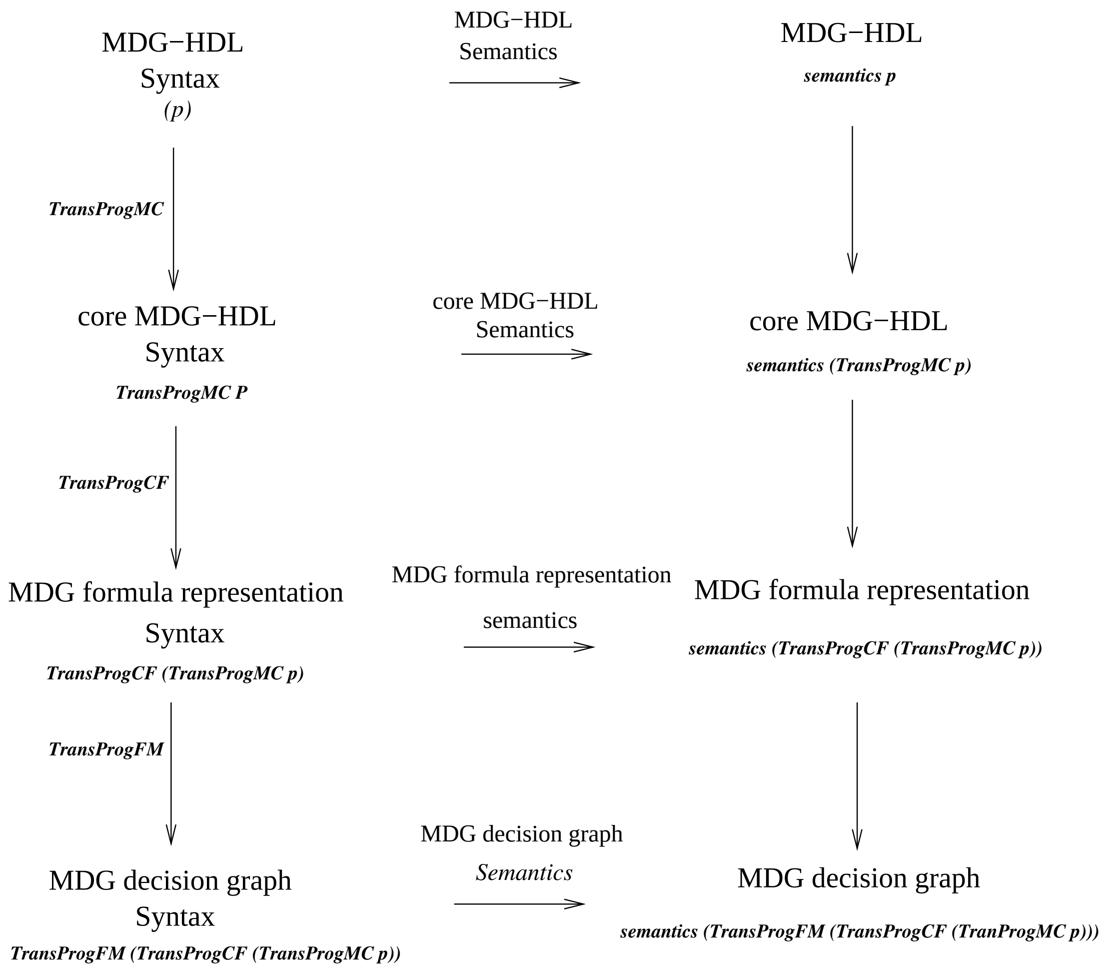


Figure 1.6: Compilation Correctness

analogous method can be used to specify and verify the translation part of the MDG system. For the translation to core MDG-HDL, the correctness theorem has the form

```
∀ p.  Semantics (p) = Semantics (TransProgMC p)
```

For the translation to the MDG formula representation, the correctness theorem has the form

```
∀ p.  Semantics (TransProgMC p) =
                Semantics (TransProgCF (TransProgMC p))
```

For the translation to the MDG decision graph, the correctness theorem has the form

```
∀ p.  Semantics (TransProgCF (TransProgMC p)) =
                Semantics (TransProgFM (TransProgCF (TransProgMC p)))
```

By combining the three correctness theorems above, we can obtain a correctness theorem. This theorem states that the semantics of the low level MDGs is equal to the semantics of the high level MDG-HDL.

```
∀ p.  Semantics (p) =
                Semantics (TransProgFM (TransProgCF (TransProgMC p)))
```

The MDG system is based on Multiway Decision Graphs which extend ROBDDs with concrete sorts, abstract sorts and uninterpreted function symbols. It can also deal with the boolean subset as other ROBDD tools do. For the sake of easily applying our method to the other decision graph based verification tools, we will define the deep embedding semantics for two different subsets of the MDG-HDL language in this thesis. Both subsets we considered in this thesis do not contain three MDG predefined components (Multiplexer, Driver and Constant) and the Transform construct used to apply functions. These components are omitted from our subsets as they have non-boolean inputs or outputs. We make the subset simple here since we want to explore the feasibility of this method.

The first subset is a boolean subset of the language which corresponds to a ROBDD system. In this subset, the table representation in the core MDG-HDL language only can be defined in terms of the corresponding boolean inputs value (true or false). We consider this subset because it corresponds to a ROBDD system. The formalization of this subset can be integrated to other ROBDD based tools with relatively small modification. For this subset, we will concentrate on verifying the first two translation steps (see (1)(2), Figure 1.5). Detail will be discussed in Chapter 3.

The second subset is an extension of the first subset. In the rest of this thesis we will call it the `extended subset`. This subset allows the program of the MDG-HDL language to contain concrete sorts. In other words, the subset we considered in this thesis is a subset language of MDG-HDL whose inputs and outputs of a `table` could be boolean sorts and concrete sorts. This is very important because this is the way the MDG system works. For coping with different types in one list, we define a new type `Mdg_Basic` in HOL. The value of the type can be either a boolean value or a string. As a result, the syntax and the semantics of this subset are more complex and the difficulty of the MDG translator verification will be increased a lot. For this subset, we will concentrate on verifying the first translation step (see (1), Figure 1.5). More detail will be discussed in Chapter 4.

1.3.2 The Importing Theorems

Generally, when we use HOL to verify a design, the design is modelled as a hierarchy structure with modules divided into submodules as shown in Figure 1.7. The submodules are repeatedly subdivided until the logic gate level is eventually

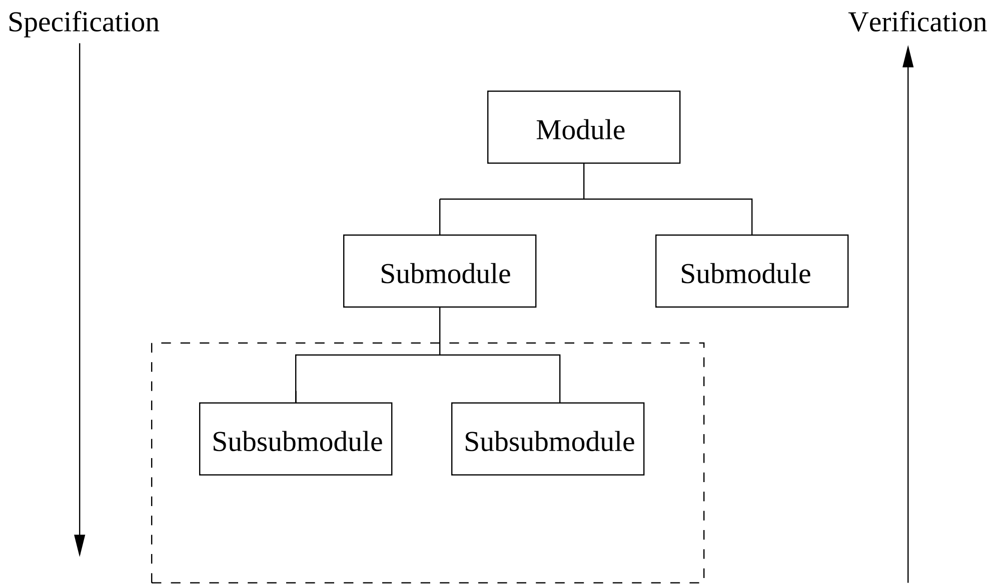


Figure 1.7: Hierarchical Verification

reached. Both the structural and the behavioral specifications of each module are given as relations in higher-order logic. The verification of each module is carried out by proving a theorem asserting that the implementation (its structure) implements (implies) the specification (its behavior). They have the very general form:

$$\texttt{implementation} \supset \texttt{specification} \tag{1.1}$$

The correctness theorem for each module states that its implementation down to the logic gate level satisfies the specification. The correctness theorem for each module can be established using the correctness theorems of its submodules. In this sense the submodule is treated as a black-box. A consequence of this is that different technologies can be used to address the correctness theorem for the submodules. In particular, we can use the MDG system instead of HOL to prove the correctness of submodules.

In order to convert the MDG verification results into HOL, we need to formalize the results of the MDG verification applications in HOL. These formalizations have different forms for the different verification applications, i.e., combinational verification gives a theorem of one form, sequential verification gives a different form and so on. However, the most natural and obvious way to formalize the MDG result does not give theorems of the form that HOL needs if we are to use traditional HOL hardware verification techniques. Therefore, we need to be able to convert the MDG results into a form that can be used. In other words, we need to prove a series of translation theorems (one for combinational verification and one for sequential verification, etc.) that state how an MDG result can be converted into the traditional HOL form:

```
Formalized MDG result  ⊃
        (implementation  ⊃ specification)
```

We have formally specified the correctness results produced by several different MDG verification applications. We have given a general importing theorem for some MDG applications. These theorems do not explicitly deal with the MDG-HDL semantics or multiway decision graphs. Rather they are given in terms of general relations on inputs and outputs. The theorems proved could be applicable for other verification systems with similar architectures based on reachability analysis or equivalence checking.

1.3.3 Combining the Translator Correctness Theorems with the Importing Theorems

In this section, we will introduce the basic idea about how to combine the translator correctness theorems with the importing theorems based on the deep embedding semantics. This combination allows the MDG results to be reasoned about in HOL in terms of the MDG input language (MDG-HDL). Ultimately in HOL we want a theorem about input language artifacts. However, the MDG verification results are obtained based on a low level data structure – a MDG representation: that is what the algorithms apply to. Therefore, the formalization of the MDG verification results in the importing theorems ought to be based on the semantics of the MDG representations. However, the theorem about the translator's correctness can be used to convert the result MDG proves about the low level representation to one about the input language (MDG-HDL). By combining the translator correctness theorems with the importation theorems, we obtain the new importing theorems which convert the low level MDG verification results into HOL to form the HOL theorems in terms of the semantics of a high level language – MDG-HDL. In other words, we are not only able to import the MDG results into HOL based on a verified MDG system, but also the MDG verification results can be converted to the theorems of HOL in terms of the semantics of MDG-HDL.

For example, if we check that three `NOT` gates are equivalent to a single `NOT gate`, the whole MDG verification process and the importing process can be illustrated in Figure 1.8. In the Figure 1.8, step (1) gives a main part of the two circuit description files (the MDG-HDL input language), which are translated into the core MDG-HDL (tabular representations) language as shown in step (2). The core MDG-HDL languages are then translated into the MDG formula representation language (step (3)). The MDG formula representation languages are further translated into the MDG decision graph language (step (4)). A set of the MDG algorithms is then applied to the MDGs in order to obtain two canonical MDGs and the MDG tool checks whether two canonical MDGs are identical and returns true or false (step (5)).

In our example the MDG tool returns true. The MDG verification results are obtained based on the low level MDGs rather than the high level language MDG-HDL. However, the translator correctness theorems state that the semantics of the low level MDG is equal to the semantics of the high level MDG-HDL (the MDG input language). By combining the translator correctness theorems, the MDG verification results can be imported into HOL based on the semantics of the MDG input language (MDG-HDL). Therefore, the traditional HOL theorem can be obtained in terms of the semantics of the MDG input language.

In this thesis, we will prove two translators for the boolean subset and one

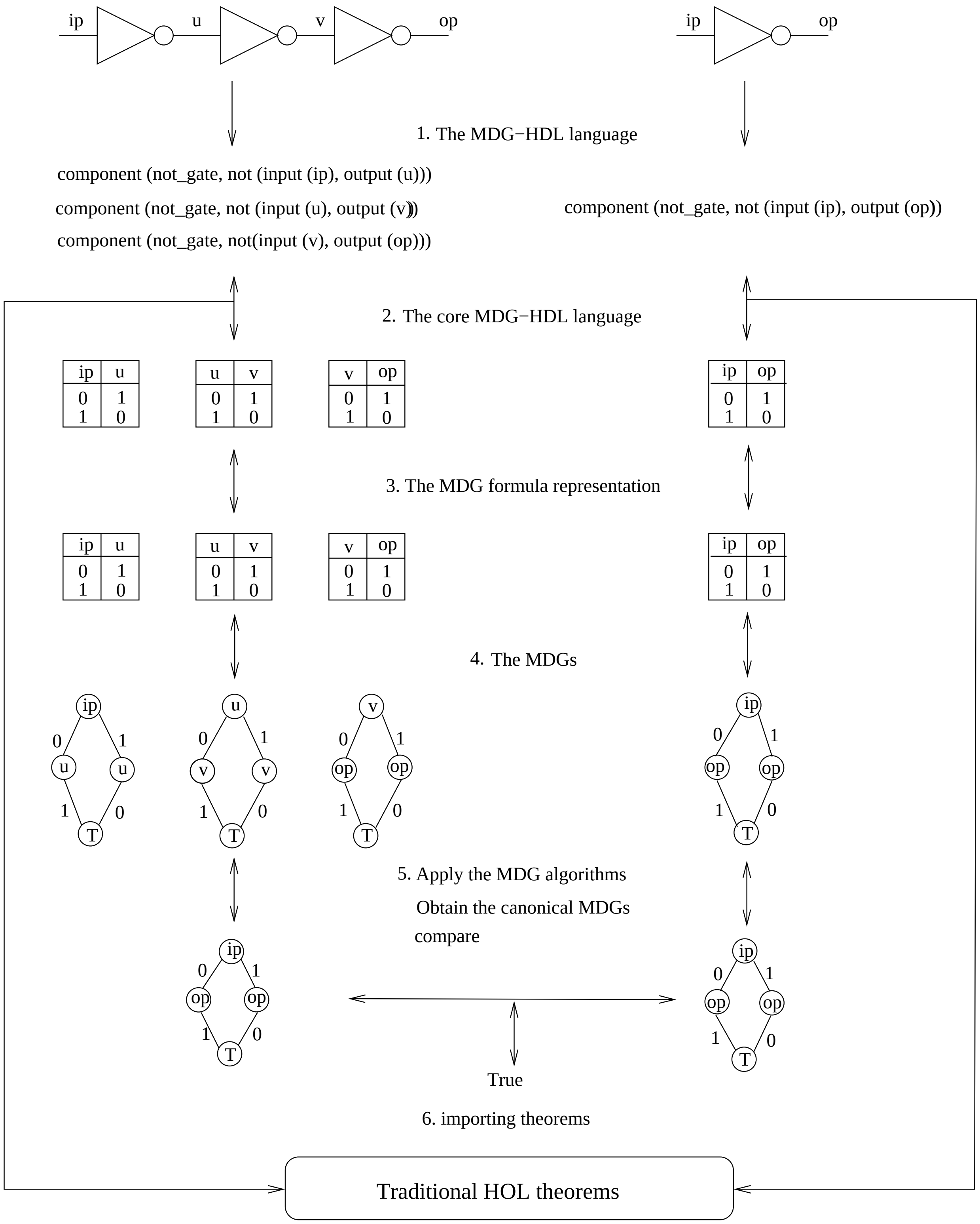


Figure 1.8: The MDG Verification Process

translator for the extended subset. In order to demonstrate the combination of the translator correctness theorems and the importing theorems, the formalization of the MDG results will be in terms of the MDG formula representation for the boolean subset and the core MDG-HDL for the extended subset. In fact, the principle is the same. Similar conversion can be done for further translators if we prove corresponding translators. By combining the translator correctness theorems with the importation theorems, we obtain the new importing theorems which convert the low level MDG verification results into HOL to form the HOL theorems in terms of the semantics of MDG-HDL. The combination also allow the additional assumption for sequential verification to be proved in terms of the semantics of MDG-HDL and the conversion theorem to be obtained in terms of the semantics of MDG-HDL.

1.3.4 Proving the Existential Theorem

In the traditional HOL hardware verification, when we prove a design, we need to prove a theorem stating that the implementation of the design implements its specification.

$$\forall \texttt{ ip op. IMPL ip op} \supset \texttt{SPEC ip op}$$

However, this representation might meet an inconsistent model that trivially satisfies any specification. This is sometimes called the "false implies anything problem" [14]. If the implementation (`IMP ip op`) of a design is false for all the inputs and outputs, then this implication is a theorem, no matter what constraint is imposed on the variables by its `SPEC ip op`. This is wrong because a theorem like this provides no meaning to ensure the correctness of the circuit. One solution to this problem is to verify a stronger consistency theorem against the implementation as suggested in [58], which has the form:

$$\forall \texttt{ ip. } \exists \texttt{ op. IMPL ip op}$$

This means that for any input `ip` there is an output `op` which is consistent with it.

On the other hand, when we formally import the MDG verification results into HOL to form the HOL theorems [80], we should prove an additional assumption against the specification. This theorem states that for all possible input traces, the behavior specification `SPEC ip op` can be satisfied for some outputs:

$\forall$ `ip.` $\exists$ `op. SPEC ip op`

This means that the machine must be able to respond to whatever inputs are given.

For ease of importing of MDG results into HOL for sequential verification and also for avoiding an inconsistent model, we will summarize a general way to prove theorem which has the form below:

$\forall$ `ip.` $\exists$ `op. C ip op`

where `C` represents any circuit, and `ip, op` represent external input and external output respectively. We called it the `existential theorem` [83]. More detail will be discussed in Chapter 7.

1.4 Outline of Thesis

The thesis is organized as follows:

In Chapter 2, we give a review of the literature most directly related to our research. We discuss embedding a hardware description language (HDL) in a proof system, previous work on verifying verification systems, an overview of compiler verification work and technologies used in the combination of different verification systems.

In Chapter 3, we investigate the verification of the translation phases of a simplified version of the MDG system (boolean subset) based on a theorem prover system (the HOL system). This can be viewed as a simple compiler correctness problem. We define a deep embedding formal semantics of the MDG-HDL language, the core MDG-HDL language and the MDG formula representation in higher order logic. A set of functions for translating the MDG-HDL subset language to their core MDG-HDL language and translating the core MDG-HDL language to their MDG formula representation language are given. The correctness theorems of the translation which quantifies over syntactic structure are verified. In particular, we demonstrate that this compiler specification preserves the correctness results produced by the MDG verification system.

In Chapter 4, we investigate the verification of the translation phases for the extended subset. We extend our formalization to accommodate a list of inputs (the first argument of the table component) with boolean sorts and concrete sorts. For this subset, we prove the first translator. We define the formal syntax and semantics of the MDG-HDL language and core MDG-HDL language. A set of functions for translating this subset language to their core MDG-HDL equivalent has then been given. The correctness theorem about the translation, which quantifies over its syntactic structure, has been proved.

In Chapter 5, we describe how to convert the MDG results into theorems for use in the HOL system. The MDG system combines a variety of different hardware verification applications including combinational verification, sequential verification, invariant checking and model checking. We give a general importing theorem for converting MDG results of the different applications (except model checking) into HOL. The theorems proved do not explicitly deal with the MDG-HDL semantics or multiway decision graphs. They are given in terms of general relations on inputs and outputs. Thus they are applicable to other verification systems with a similar architecture.

In Chapter 6, we show how to combine the translator correctness theorems with the importing theorems for two subsets. This combination allows the MDG results to be reasoned about in HOL in terms of the MDG input language (MDG-HDL). The two different MDG verification applications have been formalized in terms of the semantics of the low level language and imported into HOL to form the HOL theorems in terms of the semantics of MDG-HDL. In other word, the low level MDG verification result has been converted into a high level form which is usable in a traditional HOL hardware verification.

In Chapter 7, we summarize a general way of proving the existential theorem for the implementation and specification of any design based on the syntax and the semantics of MDG-HDL. This theorem is needed when importing the MDG sequential verification result into HOL and avoiding an inconsistent model be produced.

In Chapter 8, we use a simple example, the verification of the correctness theorem and usability theorems for a vending machine, to demonstrate the feasibility of our approach. We have verified the correctness of the vending machine in MDG. This has been imported into HOL to form the HOL theorem. We have then proved a usability theorem about a specification of the vending machine in HOL. By combining the imported theorem and specification based usability theorem, we obtain a usability theorem about the vending machine implementation.

In Chapter 9, we conclude the thesis and indicate the future work.

Summary

This chapter has motivated our emphasis on dependability of the hybrid system, and situated our approach which aims to import the MDG results into HOL in a trusted way. It also has indicated that we are concerned with how great a degree of trust the MDG system has, how to formally justify the conversion of the MDG results into the traditional HOL hardware verification theorems and how to formally link two systems in a natural way. This chapter has pointed out that the deep embedding semantics play a very important role in our research. On the one hand, the deep embedding semantics could be used to verify the correctness of aspects of the MDG system using the HOL system. On the other hand, based on the verified MDG system, the deep embedding semantics is used to combine the translator correctness theorems with the importing theorems, allowing the MDG results to be reasoned about in HOL naturally.

Chapter 2

Literature Review

Combining theorem proving systems with symbolic state enumeration systems opens a way for theorem proving systems to be applied more widely to the real world. Many researchers are working in these areas to contribute their ideas and approaches. In this thesis, we will focus on the verification of a symbolic state enumeration system (the MDG system) and provide a theoretical underpinning to the formal linkage of a symbolic state enumeration system and a theorem proving system (MDG and HOL). We first verify the correctness of translators of the MDG system by using the HOL system. This can be viewed as a simple compiler correctness problem. We next prove theorems that formally convert the MDG verification results of the MDG different applications into the traditional HOL hardware verification theorems in the style of Gordon [35]. By combining the translator correctness theorems with the importing theorems, the MDG verification results can be imported into HOL in terms of MDG-HDL. Our work is concerned with embedding a hardware description language (HDL) in a proof system, verifying verification systems, compiler verification and trusting combined systems. This chapter gives a literature review which is related to our research and divided into the corresponding subsections listed below:

- We briefly introduce embedding an HDL in a proof system.

- We discuss previous work on verifying verification systems.
- An overview of compiler verification work is given.
- We review the different technologies that have been used to combine the theorem proving systems with other systems and talk about the combined approaches and the degree of trust of the system. We then propose our own ideas.

2.1 Semantic Embedding

Semantic embedding is an approach to defining precise semantics of HDLs inside the logic so as to support the use of HDLs within a general theorem proving environment. Many researchers are aiming to find a tractable semantics for the hardware description languages such as VHDL [64]. For example, Reetz and Kropf [55] defined the semantics of a significant subset of VHDL in HOL to formalize a compiler generator. Gordon [31] defined three different semantics (event semantics, trace semantics and cycle semantic) for a subset of VHDL for use in the different applications.

There are two ways to represent the semantics of HDLs inside logic: deep embedding and shallow embedding. With a deep embedding, a type *syn*, is defined inside the logic to represent HDL texts. A type, *sem*, that represents the semantics is also defined, and then a semantic function, meaning:$syn \rightarrow sem$, is defined, by structural induction over *syn* [33] [32] [7]. With a shallow embedding there is no type *syn* or semantics function inside the logic. Instead a parser is used to translate HDL texts directly into terms of the logic. Each of these has advantages and disadvantages. The advantage of deep embedding is that it allows reasoning about classes of programs and so about the general properties of the programs. However, setting up types of abstract syntax and semantics is much work. The advantage of shallow embedding is that this work is avoided, because the process of assigning meaning to the texts does not have to be encoded as a function inside the logic.

A meta-language program can easily compute differently typed terms for different HDL texts.

Brock and Hunt [10] described a simple hardware description language in the Boyer-Moore theorem prover. It lacks delays and does not permit recursion: it thus deals with combinational logic only. However, this is the earliest research known to us which defines a deep embedding operational semantics for an HDL in a proof system. In their work, circuits were represented as list constants, which were interpreted by a semantics function. This semantics function traversed valid abstract syntax categories. The circuit descriptions were hierarchically composed. A well-formed predicate was defined to check that these definitions are purely combinational.

Melham [58] deeply embedded a denotational semantics of a CMOS circuit inside the HOL system, which is an ideal example for getting the general idea about deep embedding. He defined an abstract data type representation of CMOS circuit descriptions. A semantics function was given in terms of the environments which mapped circuits to a formula describing their switch-level behavior. The environment with the type $:string \rightarrow bool$ mapped strings *string*, denoting wire names, to their values.

Boulton et al [7] embedded semantics of three different hardware description languages in higher-order logic (HOL-ELLA, HOL-SILAGE, HOL-VHDL). Both the HOL-ELLA and the HOL-SILAGE projects used shallow embedding. The HOL-VHDL project used deep embedding. In their paper, they compared the two approaches used in three different projects and summarized the benefits of the general technique of embedding a conventional notation in a mechanized formal system and indicated that embedding the HDL semantics allows the practical tool to act directly on logic representations and thereby the designs can be reasoned about in a proof system.

Goossens [30] investigated the integration of HDLs and automated proof systems.

His aim was to clarify the semantics of the particular HDL and to present a more standard interface to formal methodologies. A formal static and dynamic operational semantics for a subset of the industrial HDL ELLA [28] were embedded within the LAMBDA proof system.

In this thesis, we deeply embed two subsets of MDG-HDL into HOL. We obtain the logic representation of each MDG-HDL program, which could be reasoned about directly into HOL. However, our aim is to verify the correctness of aspects of the MDG system by using the HOL system and to provide a formal linkage between the MDG system and the HOL system in terms of the deep embedding semantics. We use the embedding semantics to prove the translation phases of the MDG system. Our semantics explicitly represent the relation with the external wires. This representation can be used in formalizing the MDG verification results and importing the MDG results into HOL naturally. We utilize this fact to allow MDG to be used when it would be easier than obtaining the result directly in HOL.

2.2 Verifying Verification Systems

Different technologies have been used to ensure the correctness of verification systems. In a sense, which method is chosen depends on the architecture of the verification system. The Edinburgh LCF [34] (Logic of Computable Functions) family of theorem provers (including HOL) uses an abstract data type (`Thm`) to represent theorems. The type checker ensures the theorems can only be constructed by applying a small number of primitive inference rules. There is no method to construct a theorem except by carrying out a proof based on the primitive inference rules and axioms. This increases the reliability of the system. For HOL, these primitive inference rules have been proved sound via a set-theoretic semantics [40]. Pottinger [68] has also proved that they are complete with respect to Henkin's general models (the methods that Henkin used to establish completeness for systems of second-order and higher-order logic). In this way if we guarantee the primitive inference rules correct then invalid theorems can be avoided.

The LCF approach also permits proofs to be recorded. Proofs can be stored in files and be represented by lists of inferences. It allows us to make use of the availability of the sequence of inferences and to check the consistency of each inference automatically. Wong [77], changed the HOL system so as to be able to record each proof and store it into proof files. He developed a proof checker to examine the correctness of the proof files – lists of inferences generated by the HOL system. The proof checker first took a proof file as an argument and then checked whether the proofs were correct or not. A log file was then produced that contained the hypotheses, lemmas used by the proof and the resulting theorem of the proof. The application of this method is significant in developing safety-critical and high-integrity systems where high confidence of correctness is required. Since a proof checker accepts the proof files containing only primitive inference rules, it may possibly be verified formally. The proof checker also provided an independent means of ensuring the validity and consistency of the proof. Some other theorem provers such as Nqthm [9] and Coq [48] already store proof trees in the system. Boyer and Dowek [8] specified and implemented a proof checker in Nqthm logic.

Is the proof checker itself correct? If the proof checker can be formally verified, it will greatly increase the confidence in the consistency of checked proofs. Since the proof checker is relatively simple, it is easier to verify than a full system. Von Wright [75] formalized the specification of a proof checker in HOL. In his work, he carefully analysed what constituted a HOL proof, formalized the syntax of the terms, types, and theorems, and defined predicates to represent the primitive inference rules. He also demonstrated how the HOL system had been used to formally verify the specification of a proof checker for higher-order logic proofs [76]. An alternative method of using refinement to verify the proof checker was also suggested by von Wright [74].

The architecture of the symbolic state enumeration based verification systems is different. In some of these systems, higher level languages such as hardware description languages are used to describe the specifications and implementations. The specifications and implementations are then translated into decision graphs. A series of algorithms in the system is used to efficiently and automatically deal with the decision graphs and obtain the correctness results. For verifying such systems, we need to prove the translators from the higher level languages into decision graphs, and to prove the algorithms correct that are used to manipulate the decision graphs.

Homeier and Martin [46] used the HOL system to verify a verification system called a verification condition generator (VCG) for a simple programming language. Since the VCG translated the annotated programs to the lists of verification conditions, the proof of the correctness of the VCG could be considered as an example of a compiler correctness problem. In other words, the proof of the correctness of the VCG can be obtained by proving a translator. The semantics of the annotated programs and verification conditions were formalized in HOL. The correctness theorems showed that the truth of the verification conditions implied the truth of the annotated programs.

Chou and Peled [17] used the HOL system to verify a non-trivial algorithm - the Partial-Order reduction technique, implemented in the protocol tool SPIN. This algorithm is used to cut down the state-space exploration performed by model checkers. They built up the groundwork of a formal infrastructure that included the mathematical support for proving various automatic verification algorithms. Their results not only gave more confidence in the algorithm but also demonstrated formal verification is a practical and useful tool.

In this thesis, we investigate the correctness of aspects of the MDG system (figure (2.1)) by using HOL. Verifying the algorithms is beyond the scope of this thesis which can be done similarly to Chou and Peled's work. We consider verifying the translation process is correct based on the deep embedding semantics. We need to verify the translator preserves the semantics of a program through the translation between languages as suggested for Homeier's work [78] [81] [79]. A difference is that Homeier used a compiler verification method to verify a software verification system.

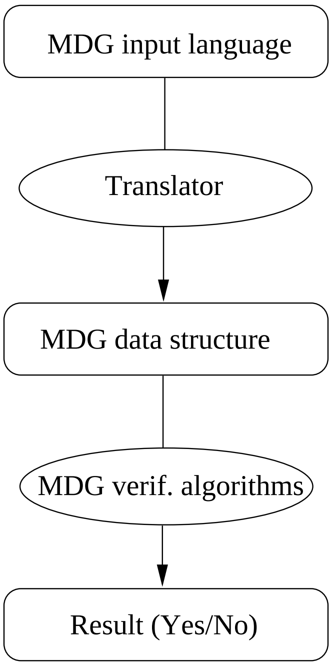


Figure 2.1: The MDG Verification System

We used a similar method to verify a hardware verification system. We consider verifying the correctness of aspects of the MDG system. In the next section, we will review previous work that has been done on the compiler correctness problem.

2.3 Verifying Compiler Correctness

The literature on compiler correctness is large. The earliest example was described more than thirty years ago [56]; this reported how McCarthy and Painter successfully verified the correctness of a simple algorithm for compiling arithmetic expressions into machine language on an ideal machine. The syntax and semantics of the source and object language were given. The compiler correctness theorem stated that the semantics of the source program preserved the semantics of the target code. Their basic idea is still being used in compiler verification.

At the same time, Burstall and Landin [13] first proposed the use of algebraic methods to verify compiler correctness. The key contribution from the algebraic approach to compiler correctness was to reject the simple function to be used as a compiler and impose structure on the program involved. Many researchers have developed this method including Morris [62] [63] and Chirica [15]. A tutorial introduction to the algebraic method was given by Collier [19]. However, the early work focused on the basic methodology rather than verifying a real language. People could not deal with the tedium of formal proof if they verified a compiler by hand.

With the development of mechanical assistance systems, researchers began to verify some simple imperative languages by using mechanical checking technology. Milner and Weyhrauch [59] used the Stanford LCF system to mechanically check the formal verification of a compiler for a simple imperative language. Cohn and Milner [18] used the Edinburgh LCF system to prove a simple parsing algorithm. In their paper, a generally mechanized method of deriving structural induction rules within the system was discussed. Chirica and Martin [16] considered the problem of proving the correctness of parsing and syntax analysis. They indicated that a compiler implementation should specify exactly how the compiler was implemented to generate the object code. The correctness of a compiler implementation is verified by comparing corresponding object programs generated by the compiler specification and implementation. However, most work including those mentioned above considered a very simple language and the target machine was idealized (no finite limitations on word size and memory size).

In 1989, Young [85] verified a code generator which was one level of a stack of verified system components by using a Boyer-Moore theorem prover (the Boyer-Moore prover is a theorem prover for a quantifier-free first-order classical logic with equality). Their source language was a subset of Gypsy [29] and the target language was the Piton [61] assembly level language. The operational semantics for a subset of Gypsy and Piton was given. Functions were implemented in the Boyer-Moore logic that translated Gypsy programs into Piton. The correctness of the translator was mechanically checked. Moore [60] verified that Piton was successfully implemented on a general purpose microprocessor (FM8502) by using the Boyer-Moore theorem prover.

Other notable work is that of Joyce [50], who described the formal specification and verification of a compiler for a very simple imperative programming language on an non-idealized target machine. The semantics for this programming language, the target machine and the compiler were all specified in higher-order logic. Inference rules of higher-order logic were used to construct a formal proof showing that compiled programs execute according to the semantics of the language. A compilation process was split into two phases for controlling the complexity of the formal proof of correctness. The first phase compiled the hierarchically structured program into a flat intermediate form. The second phase compiled the intermediate form into target machine code.

At the same time, Gordon [37] did the original work of constructing within HOL a framework for proving the correctness of a program. He used a shallow embedding [7] (i.e. only the semantics is represented in the HOL logic) to embed the program logic in the HOL logic. HOL is a foundational system which means that one can define new constants in a way that does not affect the logical consistency of the system. In other words, thus means the embedding of a language can be obtained by using constant definitions rather than by introducing arbitrary axioms to describe the semantics.

Curzon [22] successfully used the HOL system to verify compilers for a subset of the structured assembly language Vista, for a real microprocessor, VIPER. The compiler correctness work was based on a general model of I/O. The verification of a generic compiler from a generic version of Vista to a generic flat assembly code had been considered. This made it possible to verify a compiler from different versions of Vista to the VIPER microprocessor or to other similar machines easily (i.e. you just need to change some basic configurations). He also combined the verified compiler with a derived programming logic so that the corresponding properties of the compiled code can be automatically derived.

Our work concernes with verifying the correctness of the translators that translates a subset of the MDG input language MDG-HDL into the low level languages. Curzon et al. [26] did some basic work which verified the MDG components library in HOL. In their paper, the semantics of the TABLE had been first formalized in HOL. The semantics of the MDG-HDL components was in the style of Gordon. They had verified the table implementations of each of the hardware components that were implemented in terms of tables in the MDG system.

The work presented in this thesis is based on previous work to verify the MDG components library in HOL [26] and builds on the work of Curzon [22] concerned with compiler verification. The source and target languages are different to his. Our source language is a netlist level hardware description language and our target language is the core MDG-HDL language and the MDG formula representation language. We only consider the correctness of a compiler specification in this thesis. We define a deep embedding formal semantics for a subset of MDG-HDL and the corresponding low level languages in higher-order logic. However, the structures of the proofs are similar and also have been mechanically checked by using the HOL system. Most importantly, we are trying to investigate and develop a method that links compiler correctness to the combination of two different verification systems (MDG and HOL), rather than just verifies the correctness of a compiler specification.

2.4 Trusting Combined Systems

Recently, researchers have paid much attention to combining theorem provers and other symbolic computation systems. Theorem provers have been linked to other theorem provers [49], to model checkers [51] [2] [39] and to computer algebra systems [42]. Methodologies for co-operation between systems are dependent on properties of the system. The motivation for combining different systems is to achieve the benefits of them both and to make the verification simpler and more effective.

A common approach to combining proof tools is to use an automated tool as an oracle to provide results to the interactive proof process. Joyce and Seger [51] presented a hybrid verification system: HOL-Voss. In their system, several predicates were defined in the HOL system, which presented a mathematical link between the specification language of the Voss system (symbolic trajectory evaluation) [44] and that of the HOL system. As a result this link caused the specification language of Voss to become a subset of the language of the HOL system. In other words, trajectory evaluation was used as a decision procedure for the HOL proof system. A HOL tactic, VOSS_TAC, which was implemented as a remote function, was written. This tactic enabled some HOL goals to be proved by calling symbolic trajectory evaluation and mirroring the results (true or false) in HOL. If it is true, then the assertion will be transformed into a HOL theorem and this theorem can be used by the HOL system to derive additional verification results. Zhu et al. [87] successfully applied HOL-Voss to the verification of the Tamarack-3 microprocessor.

In 1995, Seger and Hazelhurst overcame some defects of the HOL-Voss system and created a new hybrid system called VossProver [43]. VossProver was implemented in `fl` (a strongly-typed functional language in the ML family [65]) in typical LCF style with an abstract datatype for theorems. Its specification language was a deep embedding in `fl` of booleans and integers and shallow embedding of tuples, lists and other features. The transition from theorem proving to model checking was done by translating the deeply embedded boolean and integer expressions into their `fl` counterparts and then evaluating the resulting `fl` expressions. A number of case studies, including the verification of a pipelined IEEE-compliant floating-point multiplier by Aagaard and Seger [3], has demonstrated the success of the approach of the system. However, the translation from the deeply embedded specification language used in the theorem proving to the normal `fl` used in the model checking was complicated. The difficulty of evaluating Boolean expressions at the `fl` prompt was a serious detraction when compared to the ease of use provided by specification in `fl`. Therefore, they wanted a proof system to use `fl` as both the specification and implementation.

In 1999, Aagaard et al developed the Forte verification system [2][1]. Forte is a combined model checking (in Voss via symbolic trajectory evaluation) and theorem proving system (ThmTac)[1]. Both specification and implementation language are `fl` which has been deeply embedded in itself so as to be lifted. In other words, the system can execute `fl` functions in Voss and reason about the behavior of `fl` functions in ThmTac. The system has successfully verified the correctness of a floating-point divider unit of an Intel IA-32 microprocessor [52].

Schneider and Kropf [70] used hardware formulas, which are higher order formulas, to express the safety and liveliness properties hierarchically. i.e. each module either consisted of a set of submodules or a basic module. These formulas could be easily translated into a model-checking problem of temporal logic. In other words, these allowed each submodule to be verified by using state enumeration techniques. Finally, the correctness results of the verified hardware could be obtained by using simple reasoning in HOL. With the same idea, an example, which could not be handled by decision procedures for temporal logic and was too expensive to use the theorem prover system, was verified easily with the combined model-checking/theorem proving approach in less than two hours. With the same idea, Schneider and Kropf [71] presented an approach for combining different proof approaches in a unifying framework to develop a hybrid system which is called C@S. This system was implemented on top of the HOL system and can be connected to the model checking system SMV (Symbolic Model Verifier) and the inductionless induction system RRL (Rewrite Rule Laboratory).

The MDG-HOL system [54] 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 of 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

[1] ThmTac is written in `fl` and is an LCF style implementation of a higher order classical logic.

design is modeled as a hierarchy structure with modules divided into submodules. The submodules are repeatedly subdivided until the design can be verified by using the MDG system. If the design of any submodule is sufficiently simple, then the hierarchical approach can be abandoned for that block and the whole module verified in one go in MDG. If submodules are all primitive components and the MDG system still cannot prove them, the HOL system can then be used to do the verification. The hybrid system is based on an embedding of the MDG hardware description language in HOL. It allows structural and behavioral specifications to be given in HOL. MDG style behavioral specifications must be used however. Essentially this means the specifications must be in the form of a finite state machine or table description. If a higher level abstraction is unavailable in MDG, a separate HOL proof must be performed to show that an MDG style specification meets this abstraction.

Gordon [38] [39] integrated the BDD based verification system BuDDY into HOL in a different way. Since "LCF-Style" general infrastructure was provided, users could implement their own BDD-based verification algorithms inside HOL by building on top of primitives provided. By implementing BDD primitives in HOL - as long as they are correct, not only could the standard state algorithms be efficiently and safely programmed in HOL, but this also made it possible to achieve the advantage of theorem reason tools and state algorithms. For example, HOL was used to formalize the QBF (Quantified Boolean Formulas) of BDDs. The formulas can be interactively simplified by using a higher-order rewriting tool such as the HOL simplifier to get simplified BDDs. A table was used to map the simplified formulas to BDDs. The BDD algorithms can also strengthen its deductive ability in this system.

Hurd [49] used a different method to combine the strengths of two theorem-prover systems. One is Gandalf which is a resolution theorem-prover for first-order classical logic with equality. The other is the HOL system. A tactic GANDALF_TAC was implemented as a remote function. It called the Gandalf system that was then run as a child process of the HOL system and mirrored the proof results to the HOL system.

Briefly, GANDALF_TAC took the input goal, converted it to a normal form, wrote it in an acceptable format, sent the string to Gandalf, parsed the Gandalf proof, translated it to a HOL proof, and proved the original goal. In this way, Gandalf's fast proof search can be used in HOL, whilst the translation into HOL ensured that the proofs were logically correct. Most importantly, in translating the Gandalf proof to HOL proof, he did not just tag the results proved in Gandalf into HOL to get HOL theorems. He wrote several functions to simulate the Gandalf proof according to the Gandalf logged file and did the proof in HOL to form the HOL theorems. As a result, the Gandalf proof results need not be tagged into HOL and the degree of trust is high. However, it is very hard to achieve a complex goal since the logged file might lose some details when the goal is very complex.

Rajan et al. [69] proposed an approach for the integration of model checking with PVS [21]: the Prototype Verification System. Harrison and Théry [42] combined the theorem prover system (HOL) and a computer algebra system (Maple). Argon and McMillan [5] attempted to use the Coq Proof Assistant to formally prove the soundness of the proof decomposition rules implemented in the SMV system. Gunter and Obradovic [41] combined a model checker (SPIN) and a theorem prover (HOL) though a language GAS (for Guarded Assignments).

A key point of combining theorem proving systems with other systems is to make the use of theorem proving systems more practical. For example, the project PROSPER [2] [27] aims to combine different interactive and automated proof tools together to deliver the benefits of them to industry. A proof management system, which is an open proof architecture, permits formal methods technology to be combined in a modular fashion. The Prosper plug-ins allow developers to add specialized verification tools (like Gandalf, Spin etc.) to the core proof engine in a relatively uniform way. In this way, different advantages of different techniques can be utilized according to the different requirement applications, whilst the translation into HOL ensures that the proofs are logically correct.

[2] The description of the project is available via the Web page http://www.dcs.gla.ac.uk/~tfm/

We have discussed many researchers using different approaches to combine different systems. Some of them, including those mentioned above, are used by the external tools as an oracle to guarantee the results provided by the external tools are theorems within the theory of the proof system. Ideally, if we could verify that the external verification tools are correct and formally convert the corresponding results into valid theorems in a current proof system, then the degree of trust of the combining system will increase a lot.

In the work presented in this thesis, we shall use this idea to provide a formal linkage between the MDG system and the HOL system. We are not using the MDG system as an oracle to then prove results, already determined, by primitive inference in HOL as MDG-HOL did, nor are we using HOL to improve the way MDG works. Furthermore, we are not just farming out general lemmas (e.g., propositional tautologies) that arise whilst verifying a particular hardware module and that can be proved more easily elsewhere. We are doing theoretical research about how to provide a formal linkage between MDG and HOL. Our formalization is defined in terms of the specification of the MDG system and MDG-HOL system. We define deep embedding formal semantics in HOL for two simplified versions of the MDG input language, to verify the correctness of translators of the MDG system in the HOL system. We also prove a series of importing theorems [80], which formally convert the formalized MDG verification results into a form usable in a traditional HOL hardware verification, i.e., the structural specification implements the behavioral specification. By combining the translator's correctness theorems with the importing theorem, the MDG verification results can be converted into HOL to form the traditional HOL theorems in terms of the semantics of MDG-HDL.

Summary

In this chapter, we have given a literature review which relates to our research including embedding an HDL in a proof system, previous work on verifying verification systems, an overview of compiler verification work and technologies used in the combination of the different verification systems. We also summarize what we did in corresponding related fields.

Chapter 3

Verifying the MDG Translators for a Boolean Subset

In this chapter, we will verify the translation phase of the MDG system as shown in step (1) (2) of Figure 1.5 for a boolean subset. Our aim is to prove the MDG translators. A standard approach for proving a translator between two languages will be used.

We will first define the syntax and the semantics of a subset MDG-HDL language, a corresponding core MDG-HDL language and the MDG formula representation language. We then define a set of functions, which translates the program from the subset MDG-HDL language to the core MDG-HDL language, and from the core MDG-HDL language to the MDG formula representation language. For each program in MDG-HDL, the compilation operators are defined as functions, which return its core MDG-HDL code and MDG formula representation code. The translation function `TransProgMC` is applied to each MDG-HDL program `p` so that the corresponding core MDG-HDL program is established and the translation function `TransProgCF` is applied to a core MDG-HDL program so that the MDG formula representation program is established. The two correctness theorems for two translation steps of this subset, which quantifies over the syntactic structure, are verified. By combining these two correctness theorems, we obtain that the semantics of the MDG-HDL program is equivalent to the semantics of the MDG formula representation program. The detail will be discussed in the following sections.

3.1 The Syntax of the MDG-HDL Language

In MDG-HDL programs, two kinds of information are provided. One is used in the MDG algorithm, the other is used in specifying the hardware. We can ignore the information which is used in the MDG algorithms when we write the syntax and semantics of programs, since this part is passed directly to the MDG algorithms and we do not consider the MDG algorithms in this thesis. Following the approach utilized in other compiler correctness work, we abstract the useful information from the MDG-HDL program and work with an abstract syntax rather than the concrete syntax of the language. It would be straightforward to write a parser that translates the MDG-HDL program into the abstract syntax.

For example, a part of the MDG-HDL file which is used to specify the hardware of three `NOT` gates and one `register` connected in series is given in Figure 3.1. The information for the algorithms is omitted.

The abstract syntax of this file is

```
PROG (EXOUT ["op"]) (EXIN ["ip"]) (INV ["u";"v";"w"])
        (JOIN (NOT "ip" "u")
              (JOIN (NOT "u" "v")
                    (JOIN (NOT "v" "w") (REG "w" "op"))))
```

where `PROG`, `EXOUT`, `EXIN`, `INV`, `JOIN`, `NOT` and `REG` are syntactic constructors of the subset of the MDG-HDL language. More details will be given later.

```
...
signal(ip,bool).
signal(op,bool).
signal(u,bool).
signal(v,bool).
signal(w,bool).

component(u_comp,not(input(ip),output(u))).
component(v_comp,not(input(u),output(v))).
component(op_comp,not(input(v),output(w))).
component(reg_comp,reg(input(w),output(op))).

outputs([op]).
...
```

Figure 3.1: The Circuit Description File of Three NOT Gates and One Register

The full abstract syntax of the subset of the MDG-HDL language is given in Appendix A. The abstract syntax of the program is represented by constructor `PROG` which is defined in terms of four arguments – an external output wires list, an external input wire list, an internal wire list and a `component term`.

```
Program ::= PROG of Exoutput => Exinput => Invariable => Mdg_Hdl
```

For example, the abstract syntax of a program of one `NOT` gate circuit is given below:

```
PROG (EXOUT ["op"]) (EXIN ["ip"]) (INV []) (NOT "ip" "op")
```

where the first argument is a list of external outputs (`["op"]`), the second is a list of external inputs (`["ip"]`) and the third is a list of internal wires (in a `NOT` gate circuit, there is no internal wire), and the final argument is the combination of the circuit components (a `NOT` gate).

In the syntax of the program, the first three arguments are variable lists. We define new HOL types `Exoutput`, `Exinput` and `Invariable` to represent the external output list, external input list and internal list respectively.

```
Exoutput ::= EXOUT of string list
Exinput ::= EXIN of string list
Invariable ::= INV of string list
```

The fourth argument (`component term`) describes how circuits are constructed from subcircuits except the hiding operations on internal wires. The hiding operations on internal wires will be defined in the semantics of the program. The `component term` could be either a predefined MDG-HDL component, an operation to set the initial value of a variable, a next state variable command, or a composition operation that denotes a circuit built up by the operation of composition. The syntax of the `component term` introduces a specially-defined recursive data type `Mdg_Hdl` to provide an explicit representation in logic of the MDG-HDL commands. We define a recursive type `Mdg_Hdl` with 33 constructors. The first 27 constructors are gates, flip-flops and registers. For example, the `component term`, '`NOT ip op`', represents a `NOT` gate with one input labeled `ip` and one output labeled `op`.

The constructor `FORK` represents the equality checker which is used to check the equality of two or more variables. The constructor `INIT` represents the initial value of a state variable. '`INIT(v,T)`' declares that the initial value of the variable `v` is true. The `SNXT` constructor maps between a state variable and a next state variable. '`SNXT v nv`' states that `nv` is the next state variable of the state variable `v`.

The `JOIN` constructor represents the composition operation. If `c1` and `c2` are two values of type `Mdg_Hdl`, then the term '`JOIN c1 c2`' represents the composition of the two terms represented by `c1` and `c2`.

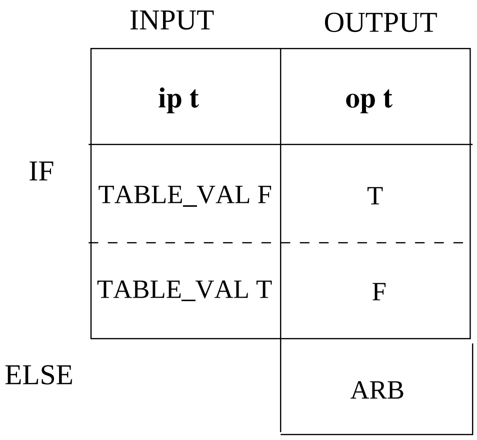


Figure 3.2: The Syntax of a NOT Gate Table

Finally, the constructor `TABLESYN` represents the syntax of the table component. It has five arguments. The first argument is a list of inputs, and the second is the single output. The third argument is a list of table rows. Each row is a list itself, giving one allocation of values to the inputs. The fourth argument is a list of output values that correspond to the values in input rows. We called the third argument an "if condition", which means if the value of input matches the corresponding row of the table then the output value will be one of the elements in the fourth argument's list. The final argument is the default value, which is taken by the output if the input values do not match any row of the third argument. We called those input values as the "else condition". The "else condition" is not listed in the third argument of constructor `TABLESYN`. For example, the abstract syntax of a `NOT` gate table is given below:

```
TABLESYN ["ip"] (NOWV "op") [[TABLE_VAL F];
                              [TABLE_VAL T]]
                                 [T; F] (DENORMAL ARB)
```

where `"ARB"` is the predefined HOL term representing an arbitrary value of a given type. Alternately, we can use a diagram to represent the abstract syntax of the `NOT` gate table, such as the one shown in Figure 3.2.

The first argument of the constructor TABLESYN is a list of inputs. In a NOT gate table, it has only one input which is "ip". The second argument is the single output "op" whose value could be either a current state variable or a next state variable. We define a new HOL type Out_Type to represent these options:

```
Out_Type ::= NOWV of string|
             NEXTV of string
```

The output in the NOT gate table is a current state variable NOWV "op". The third argument lists all the "if condition". In a NOT gate, the "if condition" is [[TABLE_VAL F], [TABLE_VAL T]]. The entries in the list can be either actual values or a special don't care marker. This is realized by defining a new type (as given in [26]).

Table_Val ::= TABLE_VAL of α | DON'T_CARE

$\vdash_{def}$ TableVal_to_Val (TABLE_VAL (v:α)) = v

The fourth argument is a list of output values that correspond to the values in input rows (the "if condition"). The final argument could be an arbitrary boolean value, a current state variable or a next state variable. Again we define a new HOL type Default_Type in terms of the type Out_Type.

```
Default_Type ::= DENORMAL of bool|
                 DEOUT of Out_Type
```

Corresponding to our NOT gate table, if the value of input is false (TABLE_VAL F from the third argument) then the value of the output is true (T from the fourth argument), if the value of input is true (TABLE_VAL T) then the value of the output is false (F), otherwise the value of the output could be an arbitrary value.

3.2 The Syntax of the Core MDG-HDL Language

The core MDG-HDL language that we translate to is a subset of the MDG-HDL language. The abstract syntax of the program is also defined in terms of four arguments – an external output wire list, an external input wire list, an internal wire list and a `core component term`. A `core component term` only consists of four constructors. i.e. `INITC` (initialise), `SNXTC` (state variable), `TABLESYNC` (table) and `JOINC` (component composition) which correspond to the constructors `INIT, SNXT, TABLESYN` and `JOIN` in MDG-HDL.

```
Mdg_Hdl_Core ::  =
       INITC of (string#bool)|
       SNXTC of string=>string|
       TABLESYNC of (string list)=> Out_Type=> ((bool Table_Val list) list)
           => (bool list)=> Default_Type|
       JOINC of Mdg_Hdl_Core=> Mdg_Hdl_Core
```

The syntax of the core MDG-HDL program is

```
Program_Core ::=
       PROGC of Exoutput => Exinput => Invariable => Mdg_Hdl_Core
```

For example, the abstract syntax of the core MDG-HDL of the three `NOT` gates and one `REGISTER` is given in Figure 3.3.

3.3 The Syntax of the MDG Formula Representation Program

The structure of the MDG formula representation program is similar to the structure of the core MDG-HDL language. It consists of four constructors. i.e. `INITF, SNXTF,`

```
PROGC (EXOUT ["op"]) (EXIN ["ip"]) (INV ["u";"v";"w"])
   JOINC (TABLESYNC ["ip"] (NOWV "u") [[TABLE_VAL F];
                                       [TABLE_VAL T]]
                                        [T; F] (DENORMAL ARB)
        JOINC (TABLESYNC ["u"] (NOWV "v") [[TABLE_VAL F];
                                           [TABLE_VAL T]]
                                            [T; F] (DENORMAL ARB)
             JOINC (TABLESYNC ["v"] (NOWV "w") [[TABLE_VAL F];
                                                [TABLE_VAL T]]
                                                [T; F] (DENORMAL ARB))))
                  (TABLESYNC ["w"] (NOWV "op") [[TABLE_VAL T];
                                                 [TABLE_VAL F]]
                                                 [T; F] (DENORMAL ARB))))
```

Figure 3.3: The Abstract Syntax of a Core MDG-HDL Program

`TABLESYNF` and `JOINF` which correspond to the constructors `INIT`, `SNXT`, `TABLESYN` and `JOIN` in MDG-HDL. A difference is that the constructor `TABLESYNF` consists of six arguments rather than five arguments. It adds one more argument which lists the input values of the "else condition". In other words, this argument lists all the possible input values whose corresponding output value is equal to the default value. This is very important because the system needs this information for building up the MDGs.

```
Mdg_Hdl_Formula :: =
       INITF of (string#bool)|
       SNXTF of string=> string|
       TABLESYNF of (string list)=> Out_Type=> ((bool Table_Val list) list)
         => (bool list)=> ((bool Table_Val list) list)=> Default_Type|
       JOINF of Mdg_Hdl_Formula=> Mdg_Hdl_Formula
```

For example, consider an AND gate table. When it represents MDG-HDL code and core MDG-HDL code, it has five arguments. It does not list input values for the "else condition" (Figure 3.4 (a)). However, when it represents the MDG formula representation, it lists all the input values including those values whose corresponding output value is equal to the default value (the else condition) (Figure 3.4 (b)).

The abstract syntax for an AND gate component in the MDG formula representation program is shown below:

```
TABLESYNF ["ip1"; "ip2"] (NOWV "op")
     [[TABLE_VAL F; DONT_CARE];
      [TABLE_VAL T; TABLE_VAL F]] [T]
          [[TABLE_VAL T; TABLE_VAL T]] (DENORMAL (BOOL F))
```

The syntax of the MDG formula representation program is defined in a very similar way

```
Program_Formula ::=
       PROGF of Exoutput => Exinput => Invariable => Mdg_Hdl_Formula
```

3.4 Translating MDG-HDL into the Core MDG-HDL Language

The first step in specifying a translator for MDG-HDL is to define a set of functions to translate the MDG-HDL program into the core MDG-HDL language. For each component in MDG-HDL, a compilation operator is defined as a set of functions, which returns its core MDG-HDL code. For example, a NOT gate is compiled as follows:

	INPUTS		OUTPUT
	ip1 t	**ip2 t**	**op t**
IF	F	*	F
	T	F	F
ELSE			T

(a) AND gate table in MDG−HDL and core MDG−HDL

	INPUTS		OUTPUT
	ip1 t	**ip2 t**	**op t**
IF	F	*	F
	T	F	F
ELSE	T	T	T

(b) AND gate in the MDG formula representation.

Figure 3.4: The Syntax of an AND Gate Table

```
⊢def TRANS_NOT (ip:string) op =
          TABLESYNC [ip] (NOWV op) [[TABLE_VAL F];
                                    [TABLE_VAL T]] [T; F] (DENORMAL ARB)
```

For the MDG-HDL `component term`, we define a function `TransGT` inductively over the syntactic structure and this function translates the MDG-HDL `component term` into the equivalent `core MDG-HDL component term`.

```
⊢def (TransGT (NOT ip op) = TRANS_NOT ip op) ∧
      ......
      (TransGT (TABLESYN y1 y2 y3 y4 y5) = TRANS_TABLE y1 y2 y3 y4 y5) ∧
      (TransGT (JOIN (code1:Mdg_Hdl) code2) =
                    JOINC (TransGT code1) (TransGT code2))
```

For the MDG-HDL program, a function `TransProgMC` is defined in terms of the function `TransGT`

```
⊢def TransProgMC (PROG exv exi inv c) = PROGC exv exi inv (TransGT c)
```

For example, the following theorem as shown in Figure 3.5, which is obtained by rewriting with the definitions, illustrates the translation of the MDG-HDL program of three `NOT` gates discussed above.

3.5 Translating the Core MDG-HDL Program into the MDG Formula Representation Program

For doing such translation, we need to specify a translator which translates the core MDG-HDL language into the MDG formula representation program. This translator consists of a set of functions.

```
⊢thm TransProgMC (PROG ["op"] ["ip"] ["v_B"; "u_B"]
         (JOIN (NOT "ip" "v_B") (SEQ (NOT "v_B" "u_B")
                                     (NOT "u_B" "op")))) =
     PROGC ["op"] ["ip"] ["v_B"; "u_B"]
      (JOINC (TABLESYNC [ip] (NOWV u_B) [[TABLE_VAL F];
                                         [TABLE_VAL T]]
                                          [T; F] (DENORMAL ARB)
             JOINC (TABLESYNC [u_B] (NOWV v_B) [[TABLE_VAL F];
                                                [TABLE_VAL T]]
                                                 [T; F] (DENORMAL ARB)
                   TABLESYNC [v_B] (NOWV op) [[TABLE_VAL F];
                                               [TABLE_VAL T]]
                                              [T; F] (DENORMAL ARB))))
```

Figure 3.5: Translating the MDG-HDL program into the Core MDG-HDL program

A `TABLE` in MDG-HDL can be used to specify "if-then-else" conditions. It only lists the input values for those "if condition"s that are true and the corresponding output value of each input value is given in the corresponding output list. For the "else condition", because the output value is the same, a default value is given as the output value. The semantics of the `TABLE` states that if the input value is equal to one of the elements that are listed in the table, the corresponding output value is in the output list, otherwise the output value is equal to the default value. However, when the MDG tool translates the core MDG-HDL into MDG, there is a compiler which automatically finds all other possible input values for the "else condition". In our translator, we have to find all the input values for the "else condition". For these input values, the output value is the default value.

For finding the input value for the "else condition", we need to find all the input values in terms of the length of the input list or the length of each element of the table first. The input values for the "else condition" can be obtained in terms of all the possible input values and the input values for the "if condition".

First of all, we begin to find out all the possible input values. Because we consider the boolean subset here, each input has two possible values (T/F). All the possible input values are determined by the length of the list. We define a function `nlists` for generating the list of enumerations of a given length.

```
⊢def (nlists 0 = [[]]) ∧
      (nlists (SUC n) = APPEND (MAP (CONS (TABLE_VAL T)) (nlists n))
                               (MAP (CONS (TABLE_VAL F)) (nlists n)))
```

For example, `SIMP_CONV list_ss [nlists_def] ``nlists (SUC (SUC (SUC 0)))'';` lists the combination of three elements list.

```
nlists (SUC (SUC (SUC 0))) =
       [[TABLE_VAL T; TABLE_VAL T; TABLE_VAL T];
        [TABLE_VAL T; TABLE_VAL T; TABLE_VAL F];
        [TABLE_VAL T; TABLE_VAL F; TABLE_VAL T];
        [TABLE_VAL T; TABLE_VAL F; TABLE_VAL F];
        [TABLE_VAL F; TABLE_VAL T; TABLE_VAL T];
        [TABLE_VAL F; TABLE_VAL T; TABLE_VAL F];
        [TABLE_VAL F; TABLE_VAL F; TABLE_VAL T];
       [TABLE_VAL F; TABLE_VAL F; TABLE_VAL F]]
```

We then need to find out all the input values which are not listed in the "if condition". We use `Table_match` to check the matching of input value to value listed in the table of the "if condition". A match occurs if either the table value is don't-care, or the value on the input is identical to the table value. If there is a match on a given row, this input value has been listed in the table. Otherwise, we must check the next row. If there is no match, this input value is not listed in the table. In other words, this input value belongs to the "else condition" and the corresponding output equals the default value. This is defined by a function `Table_match_Lists`.

```
⊢def (Table_match_Lists inputs [] = F ) ∧
     (Table_match_Lists inputs (CONS v vs) =
         (Table_match inputs v) ∨ (Table_match_Lists inputs vs))
```

We need to check whether all the possible input values are in the "if condition" or the "else condition". This is implemented by function `Path_Check`. It obtains all the input value lists for the "else condition".

```
⊢def (Path_Check [] V_outs = []) ∧
     (Path_Check (CONS ip ips) V_outs =
         if (~(Table_match_List (MAP TableVal_to_Val ip) V_outs))
         then CONS ip (Path_Check ips V_outs)
         else (Path_Check ips V_outs))
```

As we mentioned before, all the combinations of a list are determined by the length of the listand the possible values of each element in the list. Since we consider a boolean subset here, all the combinations of a list are determined by its length (the length of input list). Therefore, the input values for the "else condition" can be defined in term of the functions `Path_Check, nlists_def` which is given below:

```
⊢def (Else_Conditions n (V_out:bool Table_Val list list) =
            ((Path_Check (nlists n) V_out)))
```

where `n` is the length of the input list. Now, we can define a function `TRANS_TABLEC` which translates the `TABLESYNC` component to the corresponding MDG formula representation.

```
⊢def TRANS_TABLEC ip op y1 y2 d =
           TABLESYNF ip op y1 y2 (Else_Conditions (LENGTH ip) y1) d
```

The function `TransCF` is defined for translating the `core MDG-HDL component term` into the `MDG formula representation term`.

```
⊢def (TransCF (INITC p) = INITF p) ∧
     (TransCF (SNXTC s s0) = SNXTF s s0) ∧
     (TransCF (TABLESYNC y1 y2 y3 y4 y5) =
                   TRANS_TABLEC y1 y2 y3 y5 y5 ∧
     (TransCF (JOINC code1 code2) =
                    JOINF (TransCF code1) (TransCF code2))
```

Finally, the core MDG-HDL program can be translated into the MDG formula representation program by the function `TransProgCF`.

```
⊢def TransProgCF (PROGC exv inv state p) =
               PROGF exv inv state (TransCF p)
```

3.6 The Semantics of the MDG-HDL Program

In this section, we will show how to define a relational semantics [36] of the MDG-HDL program for this subset. First of all, the semantics of the MDG-HDL program is defined in terms of an environment [58] [57]. An environment is a function that has type :string $\rightarrow\delta$. This function maps a variable name (modeled by strings) to the value of that variable. In our language, the environment `env` is for state variables and signals. Its value is a history function and has a type :`num`$\rightarrow$`bool`, which represents functions from time (natural numbers) to the value at that time.

A semantic function `SemProgram` for MDG-HDL programs is defined in terms of the semantics of the MDG-HDL `component term` (`SemMdghdl`). For each component in the MDG-HDL component library, we define a specific semantic function. The semantics of the MDG-HDL `component term` (`SemMdghdl`) is defined based on the semantic functions of each component. In the rest of this section, we will first define the semantic functions for each component in the MDG-HDL component library. We then define the semantics of the MDG-HDL `component term` (`SemMdghdl`). Finally, we will define the semantics of the MDG-HDL program (`SemProgram`).

We first define the semantic function for each component. The first 27 primitive components of the MDG-HDL component are mainly logic gates and flip-flops. The traditional hardware semantics can be given [35]. The semantics of these components are relations between the input values and the output values. For example, the `NOT` gate can be expressed by

$\vdash_{def}$ `SEM_NOT ip op = (`$\forall$` t.  op t = `$\sim$` (ip t))`

The semantics of `FORK` represents the equality of two state variables. On each cycle, the output value '`op`' and input value '`ip`' are identical at that time.

$\vdash_{def}$ `SEM_FORK ip op = (`$\forall$` t.  op t = ip t)`

The constructor `INIT` has two arguments. They are represented as a pair whose first component `(FST y)` is a state variable and whose second component `(SND y)` is a boolean value. The semantics of `INIT` assigns an initial value (at time zero) to the value of the variable.

$\vdash_{def}$ `SEM_INIT (y:(num->bool)#bool) = ((FST y) 0 = SND y)`

The semantics of `SNXT` represents a relation between a state variable `y` and a next state variable `ny`. It declares that the next state variable of `y` is `ny`. In other words, the value of the variable `y` at the time `t` is equal to the value of the variable `ny` at the following time.

```
⊢def SEM_SNXT ny y = (∀ t.  ny (t+1) = y t)
```

The semantics of the table was initially given by Curzon et al [26]. Since we need to use the induction theorem, we adapt their `table` definition for adding one more base case. In their definition, they define a predicate `Table_match` to check if the input values match the table values.

```
⊢def (Table_match inputs [] t = T) ∧
     (Table_match inputs (CONS v vs) t =
        (((HD (inputs) t) = TableVal_to_Val v) ∨ (v = DON'T_CARE)) ∧
         (Table_match (TL inputs) vs t))
```

The function `table` is defined in terms of `Table_match`. It has five arguments. The first argument is a list of the inputs, the second is the single output, the third is a list of table rows. Each row is a list itself, giving one allocation of values to the inputs. The fourth argument is a list of output values. Each is the value on the output when the inputs have the values in the corresponding row. The final argument is the default value, taken by the output if the input values do not match any row. It checks if there is a match on each row. If there is, the output has the corresponding value. Otherwise, the output equals the default value. Since the third and fourth argument are lists, they may have unequal lengths. When either list is empty, the output value equals the default value.

```
⊢def (table ip op [] V_out default t = (op t = default t)) ∧
     (table ip op vs [] default t = (op t = default t) ) ∧
     (table ip op (CONS v vs) V_out default t =
             ( if (Table_match ip v t)
               then (op t = (HD V_out) t)
               else (table ip op vs (TL V_out) default t)))
```

The above definition refers to the time of interest, `t`. Function `TABLE` defines a given table which will relate a given input to a given output if the table relation is true at all time.

```
⊢def TABLE ip op V_outs V_out default =
          ∀ t.  table ip op V_outs V_out default t
```

As we mentioned before, the second argument of the table is the single output. Its output could be either a current state variable or a next state variable. We define a new HOL type `Out_Type` to represent these options. The final argument is the default value, which is taken by the output if the input values do not match any row. The default value could be an arbitrary value, a current state variable or a next state variable. We also define a new HOL type `Default_Type` in terms of the type `Out_Type`. We define two functions `SEM_OUTVAR` and `SEM_DEFAULTVAR`, in order to access the corresponding values.

```
⊢def (Sem_Outvar (NOWV y) env = (env y)) ∧
     (Sem_Outvar (NEXTV y) env = (env y) o NEXT)

⊢def (Sem_Defaultvar (DENORMAL y) env = (λ(t:num).  y)) ∧
     (Sem_Defaultvar (DEOUT x) env = (Sem_Outvar x env))
```

The values give in the list of the outputs are signals, which are functions from time to a value. Function `CONST_TO_FUNCT` is used to lift the constant list to a signal list.

```
⊢def (CONST_TO_FUNCT [c] = [λ(t:num).  c]) ∧
     (CONST_TO_FUNCT (CONS v vl) =
                    CONS (λ(t:num).  v) (CONST_TO_FUNCT vl))
```

Now, the semantics of the MDG-HDL `component term (SemMdghdl)` can be defined in terms of functions that we defined above as shown below.

```
⊢def (SemMdghdl (NOT ip op) env = SEM_NOT (env ip) (env op) ) ∧
     ......
     (SemMdghdl (TABLESYN y1 y2 y3 y4 y5) env =
              TABLE (MAP env y1) (SEM_OUTVAR y2 env) y3
                    (CONST_TO_FUNCT y4) (SEM_DEFAULTVAR y5 env)) ∧
     (SemMdghdl (JOIN code1 code2) env =
             ((SemMdghdl code1 env) ∧ (SemMdghdl code2 env)))
```

From the definition of `SemMdghdl` we know that the semantics of `TABLESYN` is defined in terms of the function `TABLE`:

```
⊢def SemMdghdl (TABLESYN ip (op:out_type) y3 y4 y5) env =
        TABLE (MAP env ip) (SEM_OUTVAR op env) y3
               (CONST_TO_FUNCT y4) (SEM_DEFAULTVAR y5 env)
```

For example, the semantics of the Table code of the `NOT` gate is

```
⊢thm SemMdghdl (TABLESYN [ip] (NOWV op) [[TABLE_VAL F];[TABLE_VAL T]]
           [T;F] (DENORMAL ARB)) env =
              TABLE (MAP env [ip]) (SEM_OUTVAR (NOWV op) env)
                    [[TABLE_VAL F];[TABLE_VAL T]]
                    (CONST_TO_FUNCT [T;F])
                    (SEM_DEFAULTVAR (DENORMAL ARB) env)
```

The semantics of sequencing (`JOIN`) is defined inductively in terms of the primary component commands. The semantics of `JOIN` is the conjunction of the corresponding semantics of each sub-command.

```
⊢def SemMdghdl (JOIN c1 c2) env =
                    ((SemMdghdl c1 env) ∧ (SemMdghdl c2 env))
```

Finally, the semantics of a full program can be defined in terms of some auxiliary functions. Firstly, the function of `Dsem_Int` is defined in terms of the semantics of the `component term (SemMdghdl)`. It uses existential quantification to hide the local variable from the environment of the circuit. It adds an extra entry to environment `env` for each internal wire. This effectively hides the internal wires in a `component term (code)`.

```
⊢def (Dsem_Int [] code env = SemMdghdl code env) ∧
      (Dsem_Int (CONS (w:string) ws) code (env:string->num->bool) =
        (∃ v.(Dsem_Int ws code (λwv.if (wv = w) then v else (env wv)))))
```

The semantics of a circuit is a relation on the external inputs and outputs. In order to explicitly represent the relation with the external wires, we define a function `Dsem_Ext`. It adds an extra entry to the environment `env` for each external wire (input or output). This function assigns all the values of external inputs or all the values of external outputs to a list `(var:(num→bool)list)`. In other words, each element in the list `var` indicates a value of an external input or a value of an external output. This function makes it possible to represent the semantics of a circuit explicitly as the relation between the external inputs and outputs.

```
⊢def (Dsem_Ext [] env (var:(num->bool)list)= env) ∧
      (Dsem_Ext (CONS (v:string) vs) env var =
           (Dsem_Ext vs (λwv.if (wv = v) then (HD var)
                                           else (env wv)) (TL var)))
```

We also define functions `SemExoutput`, `SemExinput` and `SemInvariable` to access values of the external output and input wires and internal wires.

$\vdash_{def}$ `SemExoutput (EXOUT x) = x`

$\vdash_{def}$ `SemExinput (EXIN x) = x`

$\vdash_{def}$ `SemInvariable (INV x) = x`

Finally, the semantics of a program `SemProgram` is based on the functions we introduced above. We first apply function `Dsem_Ext` to the external inputs, which adds an entry to the environment for all external inputs and assigns the value of each external input to an element of a list `ip`. We then apply the function `Dsem_Ext` to the external outputs. Similarly, this adds an entry to the environment for all external outputs and assigns the value of each external output to an element of a list `op`. Finally, the function `Dsem_Int` gives the semantics of the circuit in terms of the semantics of the `component term (SemMdghdl)` and uses existential quantification to hide the local variable from the environment of the circuit.

$\vdash_{def}$
```
SemProgram (PROG exoutput exinput inv code) ip op =
          let env1 = Dsem_Ext (SemExinput exinput) EmptyEnv ip
          in
              let env2 = Dsem_Ext (SemExoutput exoutput) env1 op
              in
                 Dsem_Int (SemInvariable inv) code env2
```

where `EmptyEnv` is the initial value of environment `env`.

The semantics of a program explicitly represents the relation between the external inputs and outputs. Our semantics is not only used to verify the correctness of the translation, but is also used to formally import the MDG results into HOL to form the HOL theorem. During the importation process, we have to formalize the different MDG applications (combinational verification, sequential verification and property checking and so on) and add some extra assumptions. All these formalizations are explicitly concerned with the external inputs and outputs. Our semantics make it possible to do so.

For example, the semantics of a circuit of three `NOT` gates and one `REGISTER` can be expressed as:

```
SemProgram (PROG (EXOUT ["op"]) (EXIN ["ip"]) (INV ["u";"v";"w"])
                (JOIN (NOT "ip" "u")
                      (JOIN (NOT "u" "v")
                            (JOIN (NOT "v" "w") (REG "w" "op"))))) ip op
```

By expanding the definitions, this circuit can actually be formalized as

```
∃ u v w.
  (∀ t.  u t = ∼HD ip t) ∧ (∀ t.  v t = ∼u t) ∧
  (∀ t.  w t = ∼v t) ∧ (∀ t.  HD op (t + 1) = w t)
```

It can be simplified further to

```
∀t.  HD op (t+1) = ˜ HD ip t
```

Obviously, the semantics of this circuit explicitly represents the relation between the external input list `ip` and output list `op` in the circuit.

3.7 The Semantics of the Core MDG-HDL Program

Similar to the last section, the semantics of the core MDG-HDL program (`SemProgram_Core`) is defined in terms of the semantics of `core component term` (`SemMdghdl_Core`) and functions `Dsem_Ext`, `Dsem_Int_Core`. Since the `core component term` only consists of four components, the semantics of it is determined in terms of its four semantic functions.

```
⊢def (SemMdghdl_Core (INITC init) env =
               SEM_INIT ((env (FST init)), (SND init) ) ∧
      (SemMdghdl_Core (SNXTC op st) env = SEM_SNXT (env op) (env st)) ∧
      (SemMdghdl_Core (TABLESYNC y1 y2 y3 y4 y5) env =
               TABLE (MAP env y1) (SEM_OUTVAR y2 env) y3
                     (CONST_TO_FUNCT y4) (SEM_DEFAULTVAR y5 env)) ∧
      (SemMdghdl_Core (JOINC code1 code2) env =
             ((SemMdghdl_Core code1 env) ∧ (SemMdghdl_Core code2 env)))
```

In the semantic function of the program (`SemProgram_Core`), function `Dsem_Ext` adds an entry to the environment for all external inputs and outputs, and assigns the value of each external input to an element of a list `ip` and each external output to an element of a list `op`. Function `Dsem_Int_Core` gives the semantics of the circuit in terms of the semantics of the `component term (SemMdghdl_Core)` and uses existential quantification to hide the local variable from the environment of the circuit.

```
⊢def SemProgram_Core (PROGC exoutput exinput inv code) ip op =
           let env1 = Dsem_Ext (SemExinput exinput) EmptyEnv ip
           in
               let env2 = Dsem_Ext (SemExoutput exoutput) env1 op
               in
                  Dsem_Int_Core (SemInvariable inv) code env2
```

3.8 The Semantics of the MDG Formula Representation Program

The semantics of the MDG formula representation program (`SemProgram_Formula`) is also defined in terms of the semantics of its `formula component term (SemMdghdl_Formula)` and functions `Dsem_Ext, Dsem_Int_Formula`. The semantics of the `formula component term (SemMdghdl_Formula)` is defined based on its component's semantic functions. Among those semantic functions, the semantic function for the constructor `TABLESYNF` is different to the semantic function for the constructor `TABLESYNC (TABLE)` in the last section.

For defining the semantic function for the constructor `TABLESYNF`, we need to define a function `Table_Formula` first. This function is defined in terms of `Table_match_List` and `table`. It checks if there is a match on the "if condition" for any input. If there is, the output has the corresponding value. Otherwise, the `Table_Formula` is the conjunction of the `Table_match_List` on the "else condition" and the output equals the default value.

```
⊢def Table_Formula inps out ift u elt default t =
       if (Table_match_List (MAP1 inps t) ift)
       then (table inps out ift u default t)
       else ((Table_match_List (MAP1 inps t) elt) ∧ (out t = default t))
```

The above definition refers to the time of interest, `t`. Function `TABLE_FORMULA` defines a given table which will relate a given input to a given output if the `Table_Formula` relation is true at all time.

```
⊢def TABLE_FORMULA ip op ift ifout elt default =
           ∀ t.  Table_Formula ip op ift ifout elt default t
```

The semantic functions for the constructors `INITF` and `SNXTF` are the same as we defined for the constructors `INIT, INITC, SNXT` and `SNXTC` in the last two sections. The semantics of the `formula component term` can be defined in terms of the above semantic functions.

```
⊢def (SemMdghdl_Formula (TABLESYNF y1 y2 y3 y4 y y5) s =
          (TABLE_FORMULA (MAP s y1) (Sem_Outvar y2 s) y3
            (CONST_TO_FUNCT y4) y (Sem_Defaultvar y5 s))) ∧
      (SemMdghdl_Formula (INITF init) s =
          SEM_INIT (s (FST init),SND init)) ∧
      (SemMdghdl_Formula(SNXTF op st) s = SEM_SNXT (s op) (s st)) ∧
      (SemMdghdl_Formula (JOINF m1 m2) s =
            (SemMdghdl_Formula m1 s ∧ SemMdghdl_Formula m2 s))
```

Finally, the semantics of the MDG formula representation program can be defined in a very similar way.

```
⊢def SemProgram_Formula (PROGCF exoutput exinput inv code) ip op =
            let env1 = Dsem_Ext (SemExinput exinput) EmptyEnv ip
            in
                let env2 = Dsem_Ext (SemExoutput exoutput) env1 op
                in
                   Dsem_Int_Formula (SemInvariable inv) code env2
```

3.9 Translator Correctness Theorems

To verify the correctness of translators as we suggested at the beginning of this section, we have to obtain two theorems that quantify over their syntactic structure, which state that the semantics of the source program is equivalent to the semantics of its translation form.

For verifying the first translator of this subset language, we have proved three theorems using HOL. The first theorem we have proved is `Component_TermC_THM`, which specifies the semantics of the `component term` is equivalent to the semantics of its `core MDG-HDL component term`.

```
⊢thm ∀  c.  SemMdghdl c env = SemMdghdl_Core (TransGT c) env
```

in which `c` represents any MDG-HDL `component term`, `TransGT` is the function which translates the MDG-HDL `component term` to its core MDG-HDL codes and `env` is the environment for variables. The correctness theorem is proved by structural induction on the syntax domain of the MDG-HDL `component term`.

The second theorem we have proved is `Circuit_DsemC_THM`, which is obtained in terms of the theorem `Component_TermC_THM`. It states that the semantics of a circuit is equivalent to the semantics of its translation form.

```
⊢thm ∀ inv c env .  Dsem_Int inv c env =
              Dsem_Int_Core inv (TransGT c) env
```

where `inv` represents the internal wires of the circuit and `c` is a sequence of the MDG-HDL components.

The third theorem is the correctness theorem of the program `ProgC_THM`, which is proved in terms of the theorems `Component_TermC_THM` and `Circuit_DsemC_THM`. The meaning of this theorem is similar to that of the theorem `Dsem_THM`, i.e., the semantics of a circuit is equivalent to the semantics of its translation form. However, the differences are that the external input list `ip` and output list `op` of the circuit are explicitly represented in the semantics of the program.

```
⊢thm ∀ exv exi inv c.
      SemProgram (PROG exv exi inv c) ip op =
          SemProgram_Core (TransProgMC (PROG exv exi inv c)) ip op  (3.1)
```

For verifying the second translator of this subset language, We need to prove another three theorems in a similar way. The first theorem we have proved is `Component_TermCF_THM`, which specifies that the semantics of the `core component term` is equivalent to the semantics of its `MDG formula component term`.

$\vdash_{thm}$ ∀c s. SemMdghdl_Core c s =
SemMdghdl_Formula (TransProgCF c) s

The second theorem is `Circuit_DsemCF_THM`, which states that the semantics of a circuit (core MDG-HDL program) is equivalent to the semantics of its translation form (MDG formula representation program).

$\vdash_{thm}$ ∀ inv c env. Dsem_Int_Core inv c env =
Dsem_Int_Formula inv (TransProgCF c) env

Similarly, the last theorem is `ProgCF_THM`, which is explicitly represented as the external input list and output list of the circuit, states that the semantics of a circuit of the core MDG-HDL program is equivalent to the semantics of its translation form (MDG formula representation program).

$\vdash_{thm}$ ∀ exv exi inv c.
SemProgram_Core (PROGC exv exi inv c) ip op =
SemProgram_Formula (TransProgCF (PROGC exv exi inv c)) ip op (3.2)

We have proved two translators are correct and obtained two correctness theorems (3.1)(3.2). By combining the above two correctness theorems, we obtain a new correctness theorem (3.3), which states that the semantics of a circuit of an MDG-HDL program is equivalent to the semantics of a corresponding MDG formula representation program.

$\vdash_{thm}$ ∀ exv exi inv c.
SemProgram (PROG exv exi inv c) ip op =
SemProgram_Formula
(TransProgCF (TransProgMC (PROG exv exi inv c))) ip op (3.3)

Summary

In this chapter, we have investigated a way to verify the correctness of aspects of a decision graph system (the MDG system) based on a theorem prover system (the HOL system). We have defined a deep embedding formal semantics for a boolean subset of MDG-HDL language, its core MDG-HDL codes and MDG formula representation language. Functions for translating the MDG-HDL subset languages to core MDG-HDL code and for translating the core MDG-HDL language to the MDG formula representation language are given. Two correctness theorems for two translators have been proved. By combining two translation correctness theorems, we obtain a new theorem states that the semantics of the MDG-HDL program is equivalent to the semantics of the MDG formula representation program. This combination allows the low level representation (the MDG formula representation language) to be converted to the high level language MDG-HDL. We will show, in Chapter 6, how such a translator correctness theorem can be combined with importing theorems.

Chapter 4

Verifying the MDG Translator for the Extended Subset

In the last chapter, we defined the syntax and the semantics of the boolean subset MDG-HDL language. We obtained a theorem (3.3), which states that the semantics of the MDG-HDL program is equivalent to the semantics of the MDG formula representation program used in the MDG implementation. However, this subset could not cope with many MDG applications. As a matter of fact, the formal logic used in MDG-HDL is a many-sorted first-order logic, which contains abstract sorts and concrete sorts. The concrete sort of boolean values is treated separately as it is predefined in MDG and used with most components. It is therefore treated as a special case. The inputs and outputs of the component `TABLE` could be different sorts. These sorts could be boolean sorts, concrete sorts and abstract sorts. In this chapter, we will extend our formalization to accommodate a list of inputs (the first argument of the table component) with boolean sorts and concrete sorts. We did not consider the abstract sort because the Montreal MDG-HOL system can only deal with the concrete sort and boolean sorts. Also the subset we consider is similar to that of BDD systems so has wide application.

In this chapter, we will verify the translation phase of the MDG system as shown in step (1) of Figure 1.5 for the extended subset. Similarly, the formal syntax and semantics of the MDG-HDL language and core MDG-HDL language of this subset will be defined. A set of functions for translating this subset language to its core MDG-HDL equivalent will then be given. The correctness theorem about the translation, which quantifies over its syntactic structure, will be proved. Before we start proving the correctness of the translation, we will introduce an example. It is a state transition diagram of the Timing block of the Fairisle ATM switch fabric [66] [26]. This example will explain why it is necessary to embed the extended subset into HOL.

4.1 State Transitions of the Fairisle Switch Fabric Timing Block

The Fairisle Switch Fabric is a real switch fabric designed and in use at University of Cambridge for multimedia applications. The Fairisle switch forms the heart of the Fairisle network. Curzon [23] formally verified this Fairisle Switch Fabric using HOL. Tahar et al [73] reverified it using MDG. The Fairisle Switch Fabric can be split into 3 sub-modules namely Acknowledgement, Arbitration and Data Switch. The Timing Block is a sub-module of the Arbitration. Pisini et al [67] verified the Timing Block using a hybrid system (HOL and MDG).

The Timing block controls the timing of the arbitration decision based on the frame start signal and the time the routing bytes arrive. Figure 4.1 shows the finite state machine of the behavior of this timing block, which is described using a state transition function and output function. The specification of the Timing block in MDG are as shown in Figure 4.2. An MDG table is used to represent the behavior of the Timing block. This MDG table is taken from [67].

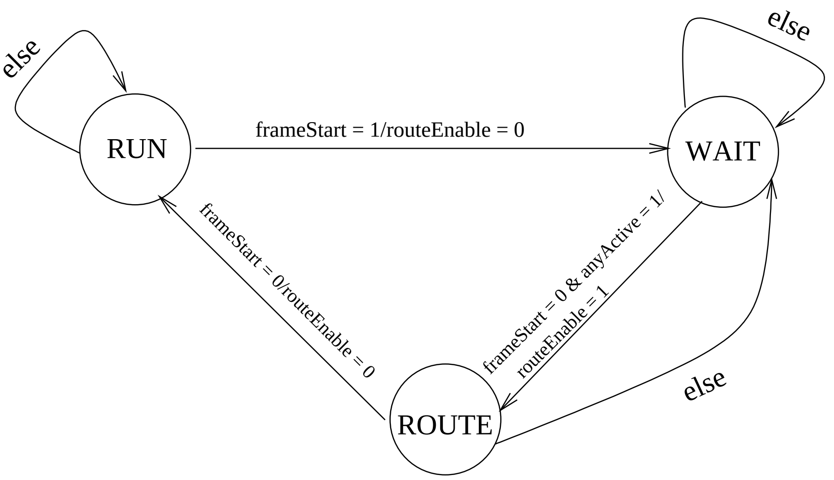


Figure 4.1: State Transitions of the Fairisle Switch Fabric Timing Block

```
table [[anyActive, frameStart, timing_state, n_timing_state],
      [*, 1, RUN, WAIT],
      [*, 0, RUN, RUN],
      [1, 0, WAIT, ROUTE],
      [*, 1, ROUTE, WAIT] | WAIT]
```

	INPUTS			OUTPUT
	anyActive	**frameStart**	**timing_state**	**n_timing_state**
IF	*	T	RUN	WAIT
	*	F	RUN	RUN
	T	F	WAIT	ROUTE
	*	T	ROUTE	WAIT
ELSE				WAIT

Figure 4.2: The Behavior of the Fairisle Switch Fabric Timing Block

In the table, `anyActive` and `frameStart` are of boolean sort, `timing_state` and `n_timing_state` are of a concrete sort with the enumeration: `RUN, WAIT, ROUTE`. In order to formalize the behavior of the Timing block, we need to redefine the definition of `TABLE` to accommodate the different sorts, as the version considered so far only allowed boolean values in the `table`. In the following section, we will redefine the syntax and semantics of the MDG-HDL language and the core MDG-HDL language to meet those requirements.

4.2 The Syntax of the MDG-HDL Language

In this section, we will define the syntax of the MDG-HDL language for the extended subset. This subset allows the program to contain concrete sorts. A concrete sort is a set of distinct constants of that sort. We use a `string` to represent them. However, the inputs and outputs of many basic components in the MDG-HDL library are of boolean value. Therefore, we use the function `Hol_datatype` to define a new type `Mdg_Basic` in HOL to meet this requirement. Since we use a boolean value to represent the inputs and outputs of some basic components and use a string to represent each element of a concrete sort (except the boolean type), this new type `Mdg_Basic` can be either a boolean value or a string. In other words, for any term with type `Mdg_Basic`, it could be a `BOOL bool` term, a `CONCRETE string` term or a base case `UNBOUND` term. In the rest of this thesis, if a variable `x` is a (`BOOL bool`) term, we say `x` is of a bool sort. If a variable `x` is a (`CONCRETE string`) term, we say `x` is of a concrete sort. If a variable `x` is a (`UNBOUND`) term, we say `x` unbound.

```
Mdg_Basic ::= UNBOUND | BOOL of bool | CONCRETE of string
```

Therefore, the common type for all the input variables of the Timing block is `Mdg_Basic`. The `anyActive` and `frameStart` are of `BOOL bool` terms, the `timing_state` and `n_timing_state` are of `CONCRETE string` terms.

The full abstract syntax of the extended subset is given in Appendix B, which is similar to that we gave before. In this version's syntax, the third argument of the constructor `TABLESYN` has the type of `((Mdg_Basic Table_Val list) list)` rather than `((bool Table_Val list) list)`. This is because each element of this argument gives one allocation of values to the inputs, while each input is of an `Mdg_Basic` term. In other words, it could be a (`BOOL bool`) term or a (`CONCRETE string`) term. Similarly, the fourth argument of it has the type of `((Mdg_Basic) list)` rather than `(bool list)`. The final argument could be an arbitrary `Mdg_Basic` value, a current state variable or a next state variable. The syntax of the table can therefore be used to formalize those designs whose MDG-HDL program contain concrete sort such as Timing block as shown below (`Timing_TABLESYN`). However, the syntax and the semantics will be complicated.

```
Timing_TABLESYN =
  (TABLESYN
   ["anyActive"; "frameStart"; "timing_state"]
   (NEXTV( "n_timing_state"))
   [[DONT_CARE; TABLE_VAL (BOOL T); TABLE_VAL (CONCRETE "RUN")];
    [DONT_CARE; TABLE_VAL (BOOL F); TABLE_VAL (CONCRETE "RUN")];
    [TABLE_VAL (BOOL T); TABLE_VAL (BOOL F); TABLE_VAL (CONCRETE "WAIT")];
    [DONT_CARE; TABLE_VAL (BOOL F); TABLE_VAL (CONCRETE "ROUTE")];
    [DONT_CARE; TABLE_VAL (BOOL F); TABLE_VAL (CONCRETE "ROUTE")]]
   [(CONCRETE "WAIT"); (CONCRETE "RUN"); (CONCRETE "ROUTE");
    (CONCRETE "RUN"); (CONCRETE "WAIT")]
   (DENORMAL (CONCRETE "WAIT")))
```

where we use `Timing_TABLESYN` to informally represent the syntax of the Fairisle Switch Fabric Timing Block. We can notice that the third argument in the table of the Timing Block contains `DONT_CARE`, boolean value (eg. `TABLE_VAL (BOOL T)`) and concrete sort value (eg. `TABLE_VAL (CONCRETE "ROUTE")`).

The abstract syntax of the program is given by the constructor `PROG`, which is similar to the `PROG` in the last chapter. It consists of an external output wire list, an external input wire list, an internal wire list and a `component term`.

```
Mdg_Program ::= PROG of Exoutput => Exinput => Invariable => Mdg_Hdl
```

For example, the syntax of the Timing block is

```
PROG (EXOUT ["n_timing_state"])
     (EXIN ["anyActive"; "frameStart"; "timing_state"])
     (INV []) (Timing_TABLESYN)
```

4.3 The Syntax of the Core MDG-HDL Language

The syntax of the core MDG-HDL language for the extended subset is similar to the syntax of the core MDG-HDL language for the boolean subset. However, their syntactic categories are different. The syntactic category for the extened subset is wider than the boolean subset, because the syntax for the extended subset can accommodate both concrete sort and boolean sort.

The abstract syntax of the program is also defined in terms of four arguments – an external output wire list, an external input wire list, an internal wire list and a `core component term`. A `core component term` only consists of four constructors. i.e. `INITC, SNXTC, TABLESYNC and JOINC`.

```
Mdg_Hdl_Core ::=
       INITC of (string#Mdg_Basic)|
       SNXTC of string=> string|
       TABLESYNC of (string list)=> Out_Type=> ((Mdg_Basic Table_Val list) list)
           => (Mdg_Basic list)=> Default_Type|
       JOINC of Mdg_Hdl_Core=>Mdg_Hdl_Core
```

The syntax of the core MDG-HDL program is

```
Mdg_Core_Program ::=
       PROGC of Exoutput => Exinput => Invariable => Mdg_Hdl_Core
```

4.4 Compiling MDG-HDL into the Core MDG-HDL Language

As we mentioned in the last chapter, we specified a translator for MDG-HDL to translate the MDG-HDL program into the core MDG-HDL language. However, the syntactic category is different to that in the last chapter.

Similarly, we first define a set of functions for each component. Thses functions apply to each component and return its core MDG-HDL code. For example, a `NOT` gate is compiled into

```
⊢def TRANS_NOT (x:string) y =
      TABLESYNC [x] (NOWV y) [[TABLE_VAL (BOOL T)];
                              [TABLE_VAL (BOOL F)]]
                              [BOOL F; BOOL T] (DENORMAL ARB)
```

We then define a function `TransGT` for the MDG-HDL `component term` inductively over the syntactic structure. This function translates the MDG-HDL `component term` into the equivalent core MDG-HDL form.

```
⊢def (TransGT (NOT ip op) = TRANS_NOT ip op) ∧
      ......
      (TransGT (TABLESYN y1 y2 y3 y4 y5) = TRANS_TABLE y1 y2 y3 y5 y5 ∧
      (TransGT (JOIN (code1:Mdg_Hdl) code2) =
                   JOINC (TransGT code1) (TransGT code2))
```

Finally, a function `TransProgMC` is defined in terms of the function `TransGT` which translates the MDG-HDL program into its core MDG-HDL program.

$\vdash_{def}$ `TransProgMC (PROG exv exi inv p) = PROGC exv exi inv (TransGT p)`

4.5 The Semantics of the MDG-HDL Program

In this section, we will define the semantics of the MDG-HDL language for the extended subset. We will first define the semantic functions for each component in the MDG-HDL component library. We then define the semantics of the MDG-HDL `component term` (`SemMdghdl`). We next define some predicates to check if all the external wires have proper values. Finally, we will define the semantics of the MDG-HDL program (`SemProgram`).

Firstly, we begin to define the semantics of the MDG-HDL components. The primitive components of the MDG-HDL `component term` are logic gates, flip-flops, table, initial value etc. The semantics of the logic gates and flip-flops are similar to the semantics we defined for the boolean subset. However, they are more complex now because we consider a different subset. The variables in this subset have different sorts. We have to define some predicates to ensure each variable does not get sort mismatched. For example, a `NOT` gate can only have boolean values. It is meaningless to have non boolean input. In other words, the type of inputs and outputs of the component in this subset is `Mdg_Basic`, we need to check if the input or output is either a `BOOL bool` term, a `CONCRETE string` term or an `UNBOUND` term for the different components and different applications. Three predicates `IS_BOOL`, `IS_CONCRETE` and `IS_UNBOUND` are defined to find out what kind of sort an `Mdg_Basic` term has.

$\vdash_{def}$ `(IS_BOOL (BOOL v) = T)` $\wedge$
`(IS_BOOL (CONCRETE u) = F)` $\wedge$
`(IS_BOOL UNBOUND = F)`

```
⊢def (IS_CONCRETE (BOOL v) = F) ∧
     (IS_CONCRETE (CONCRETE u) = T) ∧
     (IS_CONCRETE UNBOUND = F)
```

```
⊢def (IS_UNBOUND (BOOL v) = F) ∧
     (IS_UNBOUND (CONCRETE u) = F) ∧
     (IS_UNBOUND UNBOUND = T)
```

The semantics of the logic gates and flip-flops are then a conjunction of the sort judgment of its inputs and outputs and a relation between the input values and the output values. For example, the `NOT` gate can be expressed by

```
⊢def SEM_NOT ip op =
     (∀ t.  IS_BOOL (x t) ∧ (IS_BOOL (y t)) ∧
         ((MDG_TO_BOOL (y t)) = (∼ MDG_TO_BOOL (x t))))
```

where predicate `IS_BOOL` is used to check if a value of `Mdg_Basic` term is `BOOL T` or `BOOL F`, and function `MDG_TO_BOOL` converts the `Mdg_Basic` terms `BOOL T` and `BOOL F` to boolean values `T` and `F`.

```
⊢def (MDG_TO_BOOL (BOOL v) = v)
```

We define the semantics of the `AND` gate in a similar way.

```
⊢def SEM_AND x1 x2 y =
     (∀ t.  (IS_BOOL (x1 t) ∧ IS_BOOL (x2 t) ∧ IS_BOOL (y t)) ∧
       ((MDG_TO_BOOL (y t)) =
           ((MDG_TO_BOOL (x1 t)) ∧ (MDG_TO_BOOL (x2 t))))
```

The semantics of other logic gates and flip-flops are also defined in a similar way. The semantics of the `TABLESYN` is extended to deal with the type `Mdg_Basic`. It is also defined in terms of the definitions of `TABLE` and `table`. The `table` function for the extended subset is defined in a similar way to the function we defined for the boolean subset, except that the type of the inputs and output are `num→Mdg_Basic` (see Figure 4.2) . In other words, for any input and output of a `table`, their values are history functions from time, a natural number, to the value an `Mdg_Basic` term at that time. An `Mdg_Basic` term could be either a (`BOOL bool`) term or a (`CONCRETE string`) term. We define predicates to check that the value of inputs and output is whether (`num→ BOOL bool`) term or (`num→ CONCRETE string`) term.

The function `TABLE` for the extended subset is slightly different. It states that at all time each input and each output of the MDG table has a proper sort (bool sort, concrete sort or is unbounded) and the relation of the table is true.

```
⊢def TABLE inps out V_outs V_out default =
     ∀t.
             SortCheck_Input inps V_outs t ∧
             SortCheck_Output out (HD V_out) t ∧
             table inps out V_outs V_out default t
```

where functions `SortCheck_Input` and `SortCheck_Output` are defined to check the sort of each input and output.

As we mentioned in section 3.6, the third argument of a `table` is a list of table rows. Each row is a list itself, giving one allocation of values to the inputs. The values in each column of the table determines the possible sorts of one input (either (`BOOL bool`) term, (`CONCRETE string`) term or don't_care). We can check the sort of each input in the corresponding elements in the table. We first check each row by defining a recursive function `SortCheck_Input1`.

```
⊢def (SortCheck_Input1 (ins:(num->Mdg_Basic) list) [] (t:num) = T) ∧
     (SortCheck_Input1 ins (CONS v vs) t =
                  (if (IS_BOOL (TableVal_to_Val v))
                   then (IS_BOOL ((HD ins) t))
                   else (if (IS_CONCRETE (TableVal_to_Val v))
                         then (IS_CONCRETE ((HD ins) t)) else T) ∧
                  (SortCheck_Input1 (TL ins) vs t)))
```

The predicate `SortCheck_Input1` checks whether each input is a bool sort or concrete sort in terms of a table row. If function `Table_to_Val` applies to an element in the table row and obtains a (`BOOL bool`) term, this input will be a (`num`→`BOOL bool`) term. If it obtains (`CONCRETE string`) term, the corresponding input will be a (`num`→`CONCRETE string`) term. If it is don't_care, it returns `T`.

The predicate `SortCheck_Input` is defined in terms of the predicate `SortCheck_Input1`. It checks whether all the inputs are bool sorts or concrete sorts.

```
⊢def (SortCheck_Input (ins:(num->Mdg_Basic) list) [] (t:num) = T) ∧
     (SortCheck_Input ins (CONS v vs) t =
             (SortCheck_Input1 ins v t) ∧ (SortCheck_Input ins vs t))
```

The fourth argument of a `table` is a list of output values. The predicate `SortCheck_Output` is defined to check whether the output is a bool sort or a concrete sort.

```
⊢def SortCheck_Outputout outval (t:num) =
         (if IS_BOOL (outval t) then IS_BOOL (out t)
                                 else IS_CONCRETE (out t))
```

The definition of other components such as `FORK` are very similar to the definition we gave for the boolean subset. The only difference is that the type of its inputs and output are (num→ BOOL bool) and (BOOL bool) terms instead of (num→bool) and bool terms.

Secondly, the semantics of the MDG-HDL component term (SemMdghdl) is defined in a very similar way except that the syntactic category is different to that of the definition in the boolean subset.

```
⊢def (SemMdghdl (NOT x y) env = SEM_NOT (env x) (env y)) ∧
     ...
     (SemMdghdl (TABLESYN y1 y2 y3 y4 y5) env =
        TABLE ((MAP env y1)) ((SEM_OUTVAR y2 env)) y3
            (CONST_TO_FUNCT y4) ((SEM_DEFAULTVAR y5 env)) envstbl) ∧
     (SemMdghdl (JOIN (code1:Mdg_Hdl) code2) env =
             SemMdghdl code1 env ∧ SemMdghdl code2 env )
```

Thirdly, we define some predicates to check that each external wire has a proper sort. The type of the inputs and outputs of any component is (num→Mdg_Basic). However, for any components, their inputs and outputs must be either (num→BOOL bool) terms, (num→CONCRETE string) terms or UNBOUND term.

For example, the input and output value of the NOT gate must be num→BOOL bool terms, which is corresponding to the boolean value. However, the type of input and output are (num→Mdg_Basic). The value of the input and output could therefore be (num→ CONCRETE string) terms. If one of the input value or output value is an external wire and a (num→ CONCRETE string) term, the semantics of the circuit will return false. If the specification of a design returns false, the correctness theorem of this design will be always true. This is because false implies anything. In other words, an inconsistent model will be produced. When we define the semantics of the program for the extended subset, we have to add assumptions so as to avoid the sort of each variable being mismatched and the inconsistent model being produced. The assumptions are to make sure each external input and output has proper sort (either (BOOL bool) term or (CONCRETE string) term).

Since we only need to judge external wires, we define `check` to check if a variable is an external wire or not.

```
⊢def (check x [] = T) ∧
      (check x (CONS l ls) = if (x = l) then F else (check x ls))
```

where `CONS l ls` lists all the internal variables.

Predicate `BOOL_NOT` is defined to make sure that if the input or output of a `NOT` gate is an external variable then it must be a (`BOOL bool`) term.

```
⊢def BOOL_NOT (x:string) (y:string) l s =
      (∀ t.  (if (check x l) then IS_BOOL(s x t) else T) ∧
             (if (check y l) then IS_BOOL(s y t) else T))
```

Predicates for checking the sort of external inputs and outputs for other logic gates and flip-flops have been defined in a very similar way.

For checking the sort of the external inputs and output for a `table`, we have to define some auxiliary functions (`Check_Input_Sort1`, `Check_Input_Sort` and `Check_Output_Sort`). The principle of the definition of those predicates is similar to the predicates `SortCheck_Input1`, `SortCheck_Input` and `SortCheck_Output`. However, a difference is that we have to check each variable to establish whether it is an external variable first. We then check the sort of each external variable in the corresponding elements in the table.

The predicate `Check_Input_Sort1` first checks whether an input of a `table` (the first argument of the `table`) is an external wire. If it is, it finds out the sort of input in terms of a table row (the third argument of the `table`). If an element in the table row is a (`BOOL bool`) term, the value of this input will be a (`num`→`BOOL bool`) term. If it is a (`CONCRETE string`) term, the corresponding input will be a (`num`→`CONCRETE string`) term. If it is don't_care, it returns `T`.

```
⊢def (Check_Input_Sort1 (ins:string list) [] s l = T) ∧
     (Check_Input_Sort1 ins (CONS v vs) s l =
       (∀ t.  (if (check (HD ins) l))
               then (if (v = DONT_CARE) then T
                    else if (IS_BOOL (TableVal_to_Val v))
                         then (IS_BOOL ((HD ins) t))
                         else if (IS_CONCRETE (TableVal_to_Val v))
                              then (IS_CONCRETE ((HD ins) t)) else T)
                else T) ∧
           (Check_Input_Sort1 (TL ins) vs s l))
```

The predicate Check_Input_Sort is defined in terms of Check_Input_Sort1. It checks the sort of all the external wires in the table.

```
⊢def (Check_Input_Sort (inputs:string list) [] s l = T) ∧
     (Check_Input_Sort inputs (CONS v vs) s l =
              (Check_Input_Sort1 inputs v s l) ∧
                   (Check_Input_Sort (inputs) vs s l))
```

The fourth argument of a table is a list of output values. Similarly, the predicate Check_Output_Sort first checks whether the output is an external wire or not. If it is, the sort of the external output is determined in terms of the output value (the fourth argument of the table).

```
⊢def Check_Output_Sort out outval s l =
         ∀t.  (if (check (Outvar_Val out))
                then (if (IS_BOOL (outval t))
                      then IS_BOOL ((Sem_Outvar out s) t)
                      else IS_CONCRETE ((Sem_Outvar out s) t))
                else T)
```

Predicate Bool_Concrete_Table is defined for checking the sort of external inputs and outputs for the TABLE component. It is in terms of the above predicates.

```
⊢def Bool_Concrete_Table inps out V_outs V_out s l =
          ((Check_Input_Sort inps V_outs s l) ∧
                    Check_Output_Sort out (HD V_out) s l)
```

The predicate Check_External_Sort is defined inductively over the syntactic structure for checking the sort of the external wires of a circuit. It is in terms of those predicates for checking the sort of each component. The definition is given below.

```
⊢def (Check_External_Sort (NOT x y) s l = BOOL_NOT x y s l) ∧
       ......
      (Check_External_Sort (TABLESYN y1 y2 y3 y4 y5) s l =
            Bool_Concrete_Table y1 y2 y3 (CONST_TO_FUNCT y4) s l) ∧
      (Check_External_Sort (SEQ (code1:Mdg_Hdl) code2) s l =
                ((Check_External_Sort code1 s l) ∧
                         (Check_External_Sort code2 s l)))
```

Finally, we define the semantics for the MDG-HDL program of this extened subset. The semantics of a program is described by SemProgram, which is defined in terms of the predicates Dsem_Ext, Dsem_Int and Check_External_Sort. The definition of the first two predicates are similar to that we defined before except that their syntactic categories are wider than before.

As we mentioned at the beginning of this section, the semantics of the program is defined in terms of the one environment. The environment maps a syntactic object to a history function (num→Mdg_Basic). We use function Dsem_Ext adding an extra entry to this environment for each external wire (input and output). A list ip is used to represent all the values of the external inputs and a list op is used to represent all the values of the external outputs. Therefore, the semantics of the program can be represented explicitly with the external inputs `ip` and outputs `op`. The function `Dsem_Int` uses existential quantification to hide the local variable from the environment. The entries for internal variables are added to the environment. The function `Check_External_Sort` make sure that the external wires do not get sort mismatched. The semantics of the MDG-HDL program is defined in terms of those functions.

```
⊢def SemProgram (PROG exoutput exinput inv c) ip op =
     let env1 = (Dsem_Ext (SemExinput exinput) EmptyEnv ip)
     in
       let env2 = Dsem_Ext (SemExoutput exoutput) env1 op
       in
     ((Check_External_Sort c env2 (SemInvariable inv)) ⊃
             Dsem_Int (SemInvariable inv) c env2)
```

Comparing this with the semantics of the MDG-HDL program for the boolean subset (section 3.6), we notice that the semantics of the MDG-HDL program for extended subset has added an additional assumption (`Check_External_Sort`). This is because the variable in this subset can be either a boolean sort or a concrete sort. The assumption makes sure that all the external variables have proper sorts.

4.6 The Semantics of the Core MDG-HDL language

For defining the semantics of the core MDG-HDL language, we need to define the semantics of the `core component term` first (`SemMdghdl_Core`). It is defined in terms of the semantic function for each component.

```
⊢def (SemMdghdl_Core (INITC init) env =
              SEM_INIT ((env (FST init)), (SND init))) ∧
     (SemMdghdl_Core (SNXTC op st) env = SEM_SNXT (env op) (env st)) ∧
     (SemMdghdl_Core (TABLESYNC y1 y2 y3 y4 y5) env =
              TABLE (MAP env y1) (SEM_OUTVAR y2 env) y3
                  (CONST_TO_FUNCT y4) (SEM_DEFAULTVAR y5 env)) ∧
     (SemMdghdl_Core (JOINC code1 code2) env =
               ((SemMdghdl_Core code1 env) ∧ (SemMdghdl_Core code2 env)))
```

where functions `SEM_INIT`, `SEM_SNXT` and `TABLE` are semantic functions for components as we defined in the last section.

The predicate `Check_External_Sort_Core` is defined in a similar way to the predicate `Check_External_Sort` we defined in the last section. It is defined inductively over the syntactic structure for checking the sort of the external wires of a circuit. It is in terms of the sort checking predicates defined in the last section. The definition is given below.

```
⊢def (Check_External_Sort_Core (INITC init) s l = BOOL_INIT init s l) ∧
     (Check_External_Sort_Core (SNXT op st) s l = BOOL_SNXT op st s l) ∧
     (Check_External_Sort_Core (TABLESYNC y1 y2 y3 y4 y5) s l =
           Bool_Concrete_Table y1 y2 y3 (CONST_TO_FUNCT y4) s l) ∧
     (Check_External_Sort_Core (SEQ code1 code2) s l =
              ((Check_External_Sort_Core code1 s l) ∧
                    (Check_External_Sort_Core code2 s l)))
```

As in the last section, for defining the semantics of the program, we need functions `Dsem_Ext`, `Dsem_Int_Core` and `Check_External_Sort_Core`. Function `Dsem_Ext` adds an entry to the environment for all external inputs and outputs, and assigns the value of each external input to an element of a list `ip` and each external output to an element of a list `op`. Function `Dsem_Int_Core` gives the semantics of the circuit in terms of the semantics of the `core component term (SemMdghdl_Core)` and uses existential quantification to hide the local variables from the environment of the circuit. The function `Check_External_Sort_Core` find the proper sort for the external wires of the circuit. The semantics of the core MDG-HDL language is defined in terms of the above functions.

$\vdash_{def}$ `SemProgram_Core (PROGC exoutput exinput inv code) ip op =`

```
          let env1 = Dsem_Ext (SemExinput exinput) EmptyEnv ip
          in
              let env2 = Dsem_Ext (SemExoutput exoutput) env1 op
             in
               ((Check_External_Sort_Core c env2 (SemInvariable inv)) ⊃
                  Dsem_Int_Core (SemInvariable inv) code env2
```

4.7 Translator Correctness Theorem

We also prove the correctness theorem for this translator. We have proved a theorem which quantifies over its syntactic structure and states that the semantics of the MDG-HDL program is equivalent to the semantics of the core MDG-HDL program used in the MDG implementation. For proving the correctness theorem `PROG_THM`, we have proved three theorems `Component_TermC_THM, Circuit_DsemC_THM` and `Check_External_Sort_THM` using HOL. The first two theorems are similar to the theorems we proved for the boolean subset except that their syntactic category is different. In this subset, the `table` can be used to formalized the design whose variables are concrete sort and boolean sort rather than just boolean sort. The third theorem states that the sort of each external wire of a circuit `c` is equivalent to the sort of the corresponding external wire in its translation form `TransGT c`. This is because the sorts of the external variables do not change after the translation.

```
∀ c s l.  Check_External_Sort c s l =
              Check_External_Sort_Core (TransGT c) s l
```

The correctness theorem of the program PROG_THM is proved in terms of the above three theorems.

$$
\begin{aligned}
&\vdash_{thm} \forall\ \texttt{exv exi inv c.}\\
&\qquad \texttt{SemProgram (PROG exv exi inv c) ip op =}\\
&\qquad\qquad \texttt{SemProgram\_Core (TransProgMC (PROG exv exi inv c)) ip op}
\end{aligned}
\tag{4.1}
$$

Summary

In this chapter, we have extended our formalization to accommodate a list of inputs of the component `table` with boolean sorts and concrete sorts. This allows our formalization to cope with many MDG applications. We have defined the syntax and the semantics for this extended subset of the MDG-HDL language and its core MDG-HDL code. Functions for translating the MDG-HDL subset languages to core MDG-HDL codes are given. The correctness theorem of the translation for this subset which quantifies over the syntactic structure is verified. Our semantics of the program is represented explicitly with the external inputs `ip` and outputs `op`. The semantic function can be used to combine the translator correctness theorem with the importing theorems in Chapter 5.

Chapter 5

Importing Theorems

Each formal hardware verification system has its own advantages and disadvantages. Many hybrid tools have been developed to reap the benefits of the different verification systems presented in Chapter 2. Normally, the verification results from one system need to be translated to another system. In other words, there is a linkage between the two systems. How can we ensure that this linkage is trusted?

Many different technologies have been used to link two different systems in a trusted way, such as the work presented in [39]& [49]. We provide another way to make the linkage more natural and trustworthy. The linkage between the two systems is based on a series of importing theorems [80], which formally convert the formalized automated verification results to a form usable in a traditional HOL hardware verification, i.e., the structural specification implements the behavioral specification.

$$\begin{aligned}&\texttt{Formalized verification result} \supset \\ &\qquad\qquad (\texttt{implementation} \supset \texttt{specification})\end{aligned} \tag{5.1}$$

The importing theorems are based on the MDG verification applications. The formalizations have different forms for the different verification applications, i.e., combinational verification gives a theorem of one form, sequential verification gives a

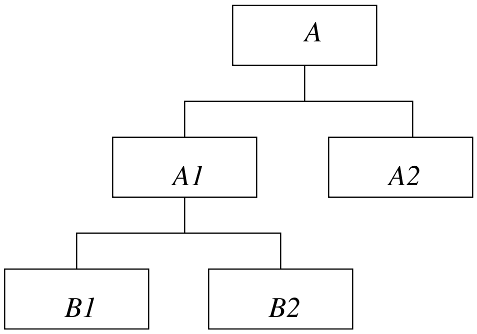


Figure 5.1: The Hierarchy of Module A

different form and so on.

To illustrate why we need a particular form of result in HOL consider the HOL verification of a system A. A theorem that the implementation satisfies its specification needs to be proved, i.e. semi formally

$$\texttt{A\_imp} \supset \texttt{A\_spec} \tag{5.2}$$

where `A_imp` and `A_spec` express the implementation and specification of system `A`, respectively. Suppose system `A` consists of two subsystems `A1` and `A2` and `A1` is further subdivided as shown in Figure 5.1. The structural specification of A will be defined by the equation:

$$\texttt{A\_imp = A1\_imp} \wedge \texttt{A2\_imp} \tag{5.3}$$

where `A1_imp` is defined in a similar way. Thus (5.2) can be rewritten to

$$\texttt{A1\_imp} \wedge \texttt{A2\_imp} \supset \texttt{A\_spec} \tag{5.4}$$

The correctness theorem of the system `A` can be proved using the correctness statements about its subsystems. In other words, we independently prove the correctness theorems:

$$\texttt{A1\_imp} \supset \texttt{A1\_spec} \tag{5.5}$$

$$\texttt{A2\_imp} \supset \texttt{A2\_spec} \tag{5.6}$$

As these are implications, to prove (5.4) it is then sufficient to prove

$$\texttt{A1\_spec} \wedge \texttt{A2\_spec} \supset \texttt{A\_spec} \tag{5.7}$$

Thus we verify `A` by independently verifying its submodules, then treating them as blackboxes using the more abstract specification of `A1` and `A2` to verify `A`.

Suppose now that `A1` is verified using MDG instead of HOL, but that we still wish to use the result in the verification of `A`. To make use of the result, we need MDG to also prove results of the form

$$\texttt{A1\_imp} \supset \texttt{A1\_spec} \tag{5.8}$$

so that the implementation can be substituted for a specification. However, results from MDG are not of this form[1]. For example, with sequential verification MDG proves a result about "reachable states" of a product machine. We need to show how such a result can be expressed as an implication about the actual hardware under consideration as above. If `A1_MDG_RESULT` is such a statement about a product machine, then we need to prove

$$\texttt{A1\_MDG\_RESULT} \supset (\texttt{A1\_imp} \supset \texttt{A1\_spec}) \tag{5.9}$$

Theorems such as this convert MDG results to the appropriate form to make the step between (5.4) and (5.7).

Ideally, we want a general theorem of this form that applies to any hardware verified using MDG's sequential verification tool. We also want similar results for the other MDG verification applications. In this chapter, we will consider each of the verification applications of the MDG system in turn, describing the conversion theorem required to convert results to a form useful within a HOL proof. Each of these theorems has been proved within the HOL system.

[1] We give details of the form of theorems that MDG does prove in the next section.

5.1 Combinational Verification

The simplest verification application of MDG is the checking of equivalence of input-output for two combinational circuits. A combinational circuit is a digital circuit without state-holding elements or feedback loops, so the output is a function of the current input. Combinational verification can also be used to compare two sequential circuits when a one-to-one correspondence between their registers exists and is known. In this situation, the output is also a function of the current input. The MDGs representing the input-output relation of each circuit are computed by a relational product algorithm to form the MDGs of the components of the circuit. Because an MDG is a canonical representation, we can check whether the two MDGs are isomorphic and so the circuits are equivalent. It is simple to formalize this in HOL. We use `M ip op` and `M'ip op` to represent the circuits (machines) being compared. `M` is a relation on input traces (given by `ip`) and output traces (given by `op`). The relation is true if `op` represents a possible output trace for the given input trace `ip` and is false otherwise. `M'` is a similar relation on inputs (`ip`) and outputs (`op`). An MDG combinational verification result can be formalized as:

$$\forall\ \texttt{ip op.}\ \ \texttt{M ip op = M' ip op} \tag{5.10}$$

It verifies that the two circuits are identical in behavior for all inputs and outputs. If `ip` and `op` are possible input and output traces for `M`, then they are also possible traces for `M'`, and vice versa. This is not in the form of an implication as described above. However, the MDG result does not need to be converted to a different form for it to be useful in a HOL hardware verification, since an equality can be used just as well as an implication.

5.2 Sequential Verification

The behavioral equivalence of two abstract state machines (Figure 5.2) is verified by checking that the machines produce the same sequence of outputs for every

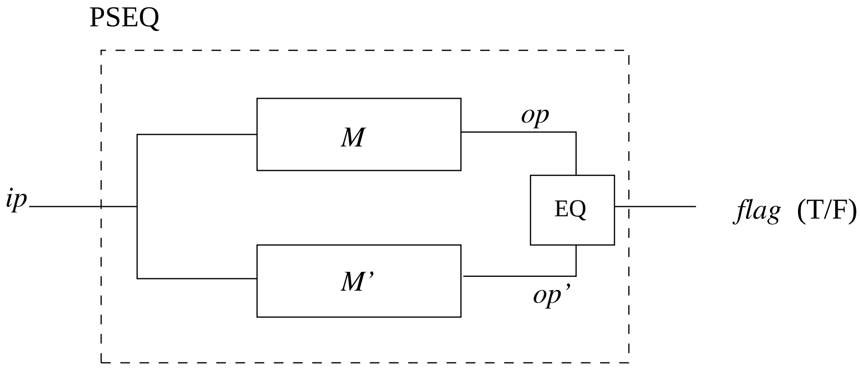


Figure 5.2: The Product Machine used in MDG Sequential Verification

sequence of inputs. The same inputs are fed to the two machines `M` and `M'` and then reachability analysis is performed on their product machine using an invariant asserting the equality of the corresponding outputs in all reachable states. This effectively introduces new "hardware" (see Figure 5.2) which we refer to here as `PSEQ` (the Product machine for SEQuential verification). `PSEQ` has the same inputs as `M` and `M'`, but has as output a single Boolean signal (`flag`). The outputs `op` and `op'` of `M` and `M'` are input into an equality checker. On each cycle, `PSEQ` outputs true if `op` and `op'` are identical at that time, and false otherwise. `PSEQ` can be formalized as

$$\begin{aligned}&\texttt{PSEQ ip flag op op' M M' =}\\ &\qquad\texttt{M ip op} \wedge \texttt{M' ip op'} \wedge \texttt{EQ op op' flag}\end{aligned} \tag{5.11}$$

Because the number of inputs and outputs of different `PSEQ` is different, we use a list to represent input `ip`, output `op`. Where `EQ` is the equality checker defined as:

$$\begin{aligned}\vdash_{def}\quad &\texttt{EQ op op' flag =}\\ &\quad(\forall\ \texttt{t. flag t = ((MAP1 op t) = (MAP1 op' t)))}\end{aligned} \tag{5.12}$$

`MAP1` is a function that applies every element of a list to the variable `t`, returning a list of the function's results:

$$\begin{aligned}\vdash_{def}\quad &\texttt{(MAP1 ([]:(}\alpha\texttt{->}\beta\texttt{) list) (t:}\alpha\texttt{) = ([]:}\beta\texttt{ list))} \wedge\\ &\texttt{(MAP1 ((x:}\alpha\texttt{->}\beta\texttt{)::l) t = (x t) :: MAP1 l t)}\end{aligned}$$

The result that MDG proves about PSEQ is that the flag output is always true, i.e., the outputs are equal for all inputs. This can be formalized as

```
∀ ip op op'.
    PSEQ ip flag op op' M M' ⊃ (∀ t.  flag t = T)
```
(5.13)

Note that this is not of the form `P_imp ⊃ P_spec`, (i.e., implementation implies specification) for M and M' but is of that form for the fictitious hardware PSEQ. To make use of such a result in a HOL hardware verification, we need to convert it to that form for M and M'. This can be done in a series of steps starting from (5.13). Expanding the definitions and rewriting with the value of flag, we obtain

```
∀ ip op op'.
    M ip op ∧ M' ip op' ⊃ (∀ t.  MAP1 op t = MAP1 op' t)
```
(5.14)

i.e., we have proved a lemma:

```
∀ M M'.
    (∀ ip op op' flag .
        PSEQ ip flag op op' M M' ⊃ ∀ t.  flag t = T) ⊃
    (∀ ip op op'.  M ip op ∧ M' ip op' ⊃
                            (∀t.  MAP1 op t = MAP1 op' t))
```
(5.15)

This is still not in an appropriate form. The theorem should also be in the form of (1.1). The machine M can be considered as the structural specification (implementation) and machine M' the behavioral specification (specification). Based on this consideration, the theorem that HOL needs is as follows:

```
∀ ip op.  M ip op ⊃ M' ip op
```
(5.16)

i.e., for all input and output traces if the relation `M ip op` is true, then the relation `M' ip op` must be true. As mentioned above, the converting theorem from MDG to HOL should be in the form of (5.1). For sequential verification the conversion theorem should be

$$(5.13) \supset (5.16).$$

To prove this, given (5.15) it is sufficient to prove

$$(5.14) \supset (5.16).$$

However, this can only be proved with an additional assumption. Namely, for all possible input traces, the behavior specification M' can be satisfied for some output (i.e., there exists at least one output for which the relation is true):

$$\forall\ \texttt{ip.}\ \ \exists\ \texttt{op'.}\ \ \texttt{M' ip op'} \tag{5.17}$$

This means that the machine must be able to respond to whatever inputs are given. This should always be true for reasonable hardware. You should not be able to give inputs which break it. For any input sequence given to this machine, at least one output will correspond. Therefore, we can actually only prove $\vdash_{thm}$ (5.13) $\wedge$ (5.17) $\supset$ (5.16),

$$\begin{array}{ll} \vdash_{thm} & \forall\ \texttt{M M'.} \\ & \quad ((\forall\ \texttt{ip op op' flag.} \\ & \qquad \texttt{PSEQ ip flag op op' M M'} \supset \forall\ \texttt{t.}\ \ \texttt{flag t = T}) \wedge \\ & \quad (\forall\ \texttt{ip.}\ \ \exists\ \texttt{op'.}\ \ \texttt{M' ip op'})) \supset \\ & \qquad\quad (\forall\ \texttt{ip op.}\ \ \texttt{M ip op} \supset \texttt{M' ip op}) \end{array} \tag{5.18}$$

With the same reasoning, the machine `M'` could have been considered as the structural specification and machine `M` could have been considered as the behavioral specification. We would then need the assumption

$$\forall\ \texttt{ip.}\ \ \exists\ \texttt{op.}\ \ \texttt{M ip op} \tag{5.19}$$

We would obtain the alternative conversion theorem (5.20)

$$\begin{array}{ll} \vdash_{thm} & \forall\ \texttt{M M'.} \\ & \quad ((\forall\ \texttt{ip op op' flag.} \\ & \qquad \texttt{PSEQ ip flag op op' M M'} \supset \forall\ \texttt{t.}\ \ \texttt{flag t = T}) \wedge \\ & \quad (\forall\ \texttt{ip.}\ \ \exists\ \texttt{op.}\ \ \texttt{M ip op})) \supset \\ & \qquad\quad (\forall\ \texttt{ip op.}\ \ \texttt{M' ip op} \supset \texttt{M ip op}) \end{array} \tag{5.20}$$

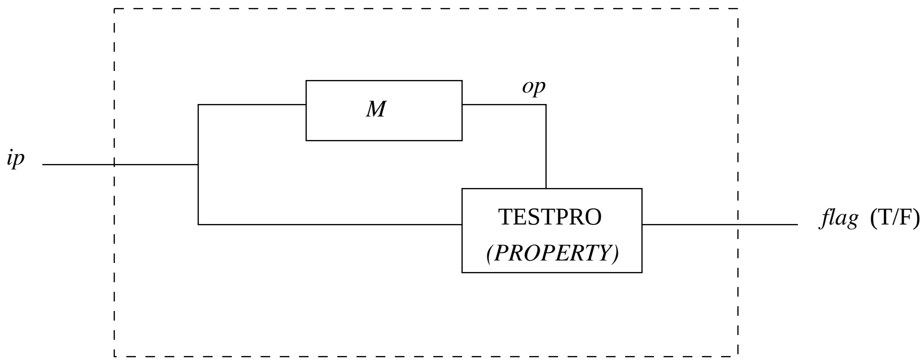


Figure 5.3: The Machine Verified in Invariant Checking

Both these theorems have been verified in HOL. As with combinational verification, the universal quantification of `M` and `M'` means the theorems can be instantiated for any hardware under consideration. The symmetry in these equations is as might be expected given the symmetry of `PSEQ`.

5.3 Invariant Checking.

Systems such as MDG also provide property/invariant checking. Invariant checking is used for verifying that a design satisfies some specific requirements. This is useful since it gives the designer confidence at low verification cost. In MDG, reachability analysis is used to explore and check that a given invariant (property) holds in all the reachable states of the sequential circuit under consideration, `M`. We consider one general form of property checking here.

As was the case for sequential verification, we introduce new "hardware" (see Figure 5.3) which we refer to as `PINV` (Product machine for INVariant checking). It consists of the original hardware and hardware representing the test property[2] wired together so that the property circuit has access to both the inputs and outputs of the circuit under test. `PINV` checks whether the outputs of the machine `M` satisfy the

[2]Invariants in MDG must be written in or converted to the same hardware description language as the actual hardware.

specific property or not. It is formalized as follows:

$$\texttt{PINV ip flag op M PROPERTY =} \\ \texttt{M ip op} \wedge \texttt{TESTPRO ip op flag PROPERTY} \tag{5.21}$$

where

$$\vdash_{def} \quad \texttt{TESTPRO ip op flag PROPERTY =} \\ (\forall\, \texttt{t. flag t = PROPERTY (MAP1 ip t) (MAP1 op t))} \tag{5.22}$$

i.e., TESTPRO is a piece of hardware which tests if its inputs and outputs satisfy some specific requirements given at each time instance by `PROPERTY`. `PROPERTY` is a relation on input and output values. Again in discussing correctness it is actually a result about this different hardware that we obtain from the property checking. The result that the property checking proves about `PINV` can be stated as:

$$\forall\, \texttt{ip flag op.} \\ \texttt{PINV ip flag op M PROPERTY} \supset \forall\, \texttt{t. flag t = T} \tag{5.23}$$

i.e., its specification is that the `flag` output should always be true. Note that this is not of the form (1.1) (i.e., implementation implies specification) for `M` but in that form for the fictitious hardware `PINV`. To make use of such a result in a HOL hardware verification we need to convert it to the form:

$$\forall\ \texttt{ip op. M ip op} \supset \forall\, \texttt{t. PROPERTY (ip t) (op t)} \tag{5.24}$$

i.e., for all input and output sequences, if the relation `M ip op` is true then the relation `PROPERTY` must be true for the input and output values at all times. In other words, the machine `M` satisfies the specific requirement $\forall$ `t. PROPERTY (ip t) (op t)`. Hence the conversion theorem for invariant checking is:

$$\vdash_{thm} \quad \forall\, \texttt{M PROPERTY.} \\ (\forall\, \texttt{ip flag op.} \\ (\texttt{PINV ip flag op M PROPERTY} \supset \forall\, \texttt{t. flag t = T})) \supset \\ (\forall\, \texttt{ip op. M ip op} \supset \\ \forall\, \texttt{t. PROPERTY (MAP1 ip t) (MAP1 op t))} \tag{5.25}$$

We have proved this general conversion theorem in HOL. Once more the theorems can be instantiated for any hardware and property under consideration.

We have looked explicitly at the MDG and HOL systems. However, the general approach could be applied to the results importation between other systems. The results could also be extended to other verification applications. Furthermore, our treatment is very general. The theorems proved do not explicitly deal with the MDG-HDL semantics or multiway decision graphs. Rather they are given in terms of general relations on inputs and outputs. Thus they are applicable to other verification systems with a similar architecture based on reachability analysis, equivalence checking and/or invariant checking. This could include a pure BDD based system.

Summary

In this chapter, we introduced how to formally specify the correctness results produced by three different hardware verification applications using HOL. We have in each case proved a general theorem that translates them into a form usable in a traditional HOL hardware verification, i.e., that the structural specification implements the behavioral specification. The first application considered was combinational verification. The next application considered was sequential verification, which checks that two abstract state machines produce the same sequence of outputs for every sequence of inputs. Finally, we considered a general form of the checking of invariant properties of a circuit.

Chapter 6

Combining the Compiler Correctness Theorems with the Importing Theorems

As we mentioned in the last chapter, the main idea of the importing theorem can be represented as below.

```
Formalized MDG result ⊃
                (implementation ⊃ specification)
```

MDG verification results are obtained by applying the MDG algorithms to MDG decision graphs. The MDG algorithms really prove properties of the low level data structures (MDGs). However, specifications and implementations are not described directly as decision graphs. A high level language, MDG-HDL, is used to specify specifications and implementations, which are translated into the multiway decision graphs (MDGs) via intermediate languages. If the MDG algorithms are correct, MDG results can be formalized in terms of the semantics of the MDG decision graphs. If the translations are correct, the semantics of the MDG decision graphs

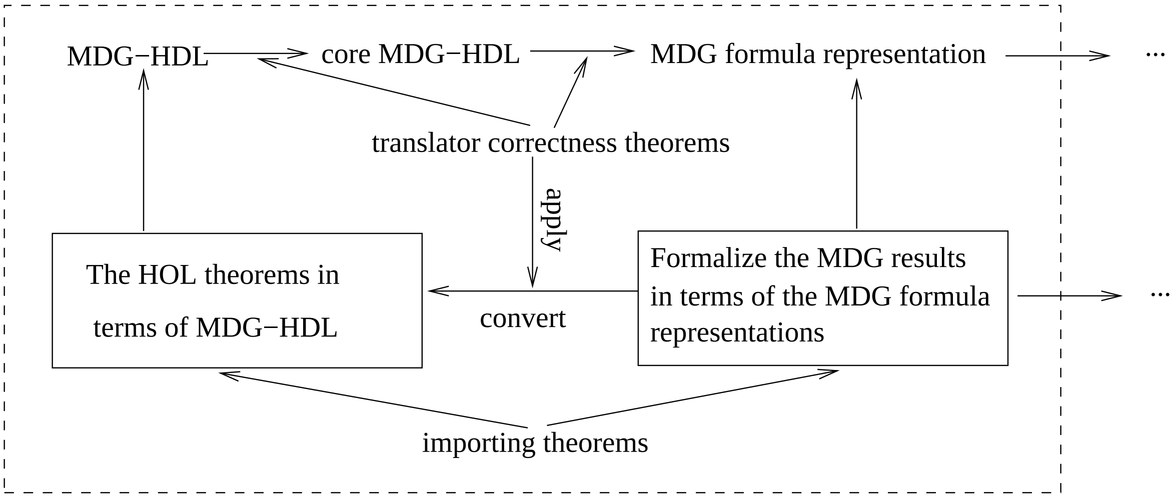


Figure 6.1: Combining the Translator Correctness Theorems with Importing Theorems for a Boolean Subset

is equal to the semantics of MDG-HDL. By combining the translator correctness theorems with the importing theorem, the MDG results can be imported into HOL to form the HOL theorems in terms of the semantics of the high level language MDG-HDL rather than in terms of the semantics of the low level language MDGs.

We have partly proved the translators for two different subsets. For the boolean subset, we have proved two translators which are correct. We have obtained a theorem which states that the semantics of the MDG-HDL program is equivalent to the semantics of the MDG formula representation program (3.3). In order to demonstrate the combination of the translator correctness theorems and the importing theorems, the formalization of the MDG results for the boolean subset will be in terms of the MDG formula representation (see Figure 6.1). In fact, the principle is the same. Similar conversion can be done for further translators if we prove corresponding translator correctness theorems. In other words, the formalization of the MDG verification results we consider in this chapter is based on the semantics of the low level MDG formula representation. However, by using the translator correctness theorems, the additional assumption can be proved in terms of the semantics of MDG-HDL and the HOL theorem we imported is in terms of the semantics of

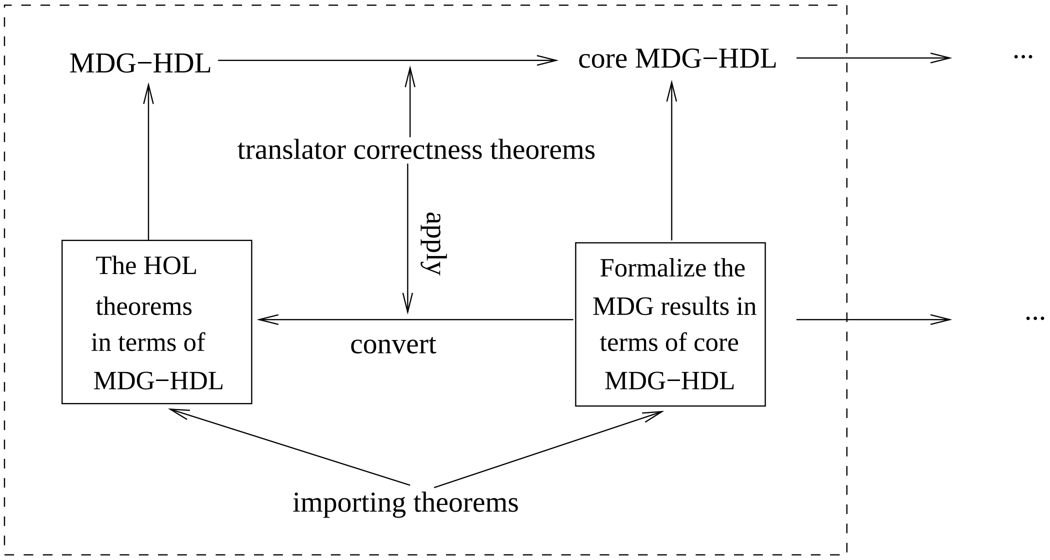


Figure 6.2: Combining the Translator Correctness Theorems with Importing Theorems for an Extended Subset

MDG-HDL.

With the same reasoning, for the extended subset, we have obtained a theorem (4.1) which states that the semantics of the MDG-HDL program is equivalent to the semantics of the core MDG-HDL program. Therefore, the formalization of the MDG results for the extended subset will be in terms of the core MDG-HDL (see Figure 6.2). By using the translator correctness theorem, the verification of the additional assumption and importation theorem are based on the semantics of the MDG input language (MDG-HDL).

The reason we are doing such a conversion is that the syntax and the semantics of a low level program are more complex and unreadable than those of a program in a high level language such as MDG-HDL. It will be more convenient, readable and direct if we prove theorems in terms of the semantics of MDG-HDL and obtain the HOL theorems in terms of the semantics of MDG-HDL. We do not take it for granted. We formally convert it from the semantics of a low level language to the semantics of a high level language in terms of the translator correctness theorems.

In this chapter, we will focus on combining the importing theorems with the translator correctness theorems. We will first instantiate the importing theorems with the syntax and semantics of a low level program for two subsets (the MDG formula representation program for the boolean subset and the core MDG-HDL program for the extended subset). We then combine the importing theorem with the translator correctness theorems and obtain the new importing theorems. The importation turns the MDG verification results based on the semantics of the low level program into HOL to form HOL theorems based on the semantics of the high level language (MDG-HDL).

6.1 Combining the Translator Correctness Theorems with the Importing Theorems for a Boolean Subset

In this section, we will firstly instantiate importing theorems with the semantics of the MDG formula representation for the combinational verification and sequential verification. By combining the translator correctness theorems, we can obtain the new importing theorems which convert the MDG verification results into HOL to form the HOL theorems in terms of MDG-HDL.

6.1.1 Combinational Verification

In combinational verification, the MDG result does not need to be converted to a different form for it to be useful in a HOL hardware verification, since an equality can be used just as well as an implication. In this situation, we just need to formalize the MDG result in terms of the semantics of the MDG formula representation. We use `C1` and `C2` to represent the abstract syntax of the circuits in MDG-HDL being compared.

The abstract syntax in the MDG formula representation will be (`TransProgCF (TransProgMC C1)`) and (`TransProgCF (TransProgMC C2)`). This is because the MDG system uses functions (`TransProgMC`) and (`TransProgCF`) which translate the MDG-HDL program to the MDG formula representation program. The semantics of the corresponding circuits is represented as (`SemProgram_Formula` (`TransProgCF (TransProgMC C1)`) `ip op`) and (`SemProgram_Formula` (`TransProgCF (TransProgMC C2)`) `ip op`). Therefore, by instantiating (`SemProgram_Formula` (`TransProgCF (TransProgMC C1)`)) and (`SemProgram_Formula` (`TransProgCF (TransProgMC C2)`)) for the machine `M` and `M'` in the combinational verification, the MDG verification result can be stated as shown below:

```
∀ ip op.
  SemProgram_Formula (TransProgCF (TransProgMC C1)) ip op =
    SemProgram_Formula (TransProgCF (TransProgMC C2)) ip op     (6.1)
```

where the formalization is in terms of the low level language (the MDG formula representation). However, as long as the MDG system returns true, this theorem can be tagged into HOL. With the help of the translator correctness theorem (3.3), we have proved a theorem `Formalize_Eqcb_Thm` (6.2) which states that the formalization of the MDG result based on a low level language is equivalent to the formalization of the MDG result based on the high level language (MDG-HDL). Therefore, the MDG verification results can be converted into HOL to form the HOL theorems in terms of the semantics of MDG-HDL.

```
⊢thm  (∀ ip op.
         SemProgram_Formula (TransProgCF (TransProgMC C1)) ip op =
           SemProgram_Formula (TransProgCF (TransProgMC C2)) ip op) =
       ∀ ip op.  SemProgram C1 ip op = SemProgram C2 ip op        (6.2)
```

Example 1. Consider the two circuits shown in Figure 6.3. Assume they have been verified to be equivalent using MDG combinational equivalence checking. We will show in the following how to convert a MDG result to a useful HOL theorem.

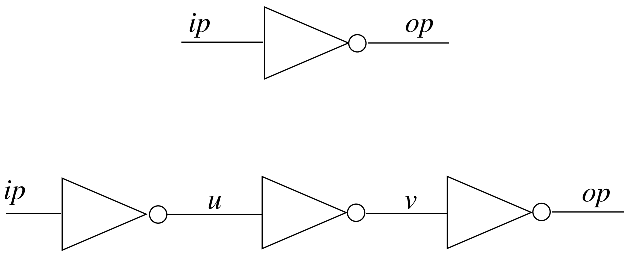


Figure 6.3: Two Equivalent Combinational Circuits

The first circuit is a single NOT gate. Its abstract syntax can be specified as:

```
NOT1 = PROG (EXOUT ["op"]) (EXIN ["ip"]) (INV [])
              (NOT "ip" "op")
```

where NOT1 is an informal abbreviation for representing the abstract syntax of this circuit. The second circuit consists of three NOT gates in series and its abstract syntax can be formalized as:

```
NOT3 = PROG (EXOUT ["op"]) (EXIN ["ip"]) (INV ["u";"v";"w"])
         (JOIN (NOT "ip" "u")
               (JOIN (NOT "u" "v")
                     (JOIN (NOT "v" "w") (REG "w" "op"))))
```

where NOT3 is an informal abbreviation for representing the abstract syntax of this circuit. The MDG verification result can be stated as

```
∀ ip op.  SemProgram (TransProgCF (TransProgMC NOT3)) ip op =
         SemProgram (TransProgCF (TransProgMC NOT1)) ip op
```

The formalization can be directly tagged into HOL to form a HOL theorem. Rewriting with the theorem Formalize_Eqcb_Thm (6.2), we obtain a new importing theorem which is in terms of the semantics of MDG-HDL.

$\vdash_{thm}$ ∀ip op. SemProgram NOT1 ip op = SemProgram NOT3 ip op

6.1.2 Sequential Verification

For sequential verification, we have obtained a general importing theorem as shown in (5.18) or (5.20). If we use `IMP` to represent an informal abbreviation of the abstract syntax of the implementation file in MDG-HDL and use `SPEC` to represent an informal abbreviation of the abstract syntax of the specification file in MDG-HDL, the corresponding informal syntax to their MDG formula representation will be (`TransProgCF (TransProgMC IMP)`) and (`TransProgCF (TransProgMC SPEC)`). The semantics of the corresponding machine can be represented as `SemProgram_Formula (TransProgCF (TransProgMC IMP)) ip op` and `SemProgram_Formula (TransProgCF (TransProgMC SPEC)) ip op`. Therefore, (`SemProgram_Formula (TransProgCF (TransProgMC IMP))`) and (`SemProgram_Formula(TransProgCF (TransProgMC SPEC))`) can be instantiated for the machine `M` and `M'` in the conversion theorem (5.18) or (5.20). Therefore, we obtain the importing theorem based on the semantics of the MDG formula representation as shown below:

```
⊢thm ∀ IMP SPEC.
      (∀ ip flag op op'.
          PSEQ ip op op' flag
                  (SemProgram_Formula (TransProgCF (TransProgMC IMP)))
                 (SemProgram_Formula (TransProgCF (TransProgMC SPEC)))
               ⊃ (∀ t.  (flag t = T))) ∧
      (∀ ip.  ∃  op'.
         SemProgram_Formula (TransProgCF (TransProgMC SPEC)) ip op') ⊃
       (∀ ip  op.
         (SemProgram_Formula (TransProgCF (TransProgMC IMP)) ip op) ⊃
         (SemProgram_Formula (TransProgCF (TransProgMC SPEC)) ip op))  (6.3)
```

When we formally import the MDG result into HOL to form the HOL theorem, we first need to formalize the MDG result in terms of the MDG formula representation and tag it into HOL.

We then need to prove an additional assumption. Namely, for all possible input traces, the behavior specification can be satisfied for some output and state traces:

$$(\forall\ \texttt{ip}.\ \ \exists\ \ \texttt{op'}.\ \ \texttt{SemProgram\_Formula (TransProgCF (TransProgMC SPEC)) ip op'}) \tag{6.4}$$

By using the translator correctness theorem (3.3), we have proved a theorem `Exist_Eq_Thm` (6.5) which states that the additional assumption based on the semantics of a low level language is equivalent to that based on the semantics of a high level language (MDG-HDL). Therefore, the additional assumption can be proved in terms of the semantics of MDG-HDL.

$$\begin{aligned} \vdash_{thm}\quad & (\forall\ \texttt{ip}.\ \ \exists\ \ \texttt{op'}.\\ & \quad \texttt{(SemProgram\_Formula (TransProgCF (TransProgMC SPEC))) ip op'}) =\\ & (\forall\ \texttt{ip}.\ \ \exists\ \ \texttt{op'}.\ \ \texttt{SemProgram SPEC ip op'}) \end{aligned} \tag{6.5}$$

Similarly, we have also proved a theorem `Imp_Eq_Thm`, which converts the trandi-tional HOL theorem (`implementation` $\supset$ `specification`) based on the semantics of the low level language to that based on the semantics of MDG-HDL.

$$\begin{aligned} \vdash_{thm}\quad & (\forall\ \texttt{ip op}.\\ & \quad \texttt{(SemProgram\_Formula (TransProgCF (TransProgMC IMP))) ip op} \supset\\ & \quad \texttt{(SemProgram\_Formula (TransProgCF (TransProgMC SPEC))) ip op}) =\\ & (\forall\ \texttt{ip op}.\ \ \texttt{(SemProgram IMP) ip op} \supset \texttt{(SemProgram SPEC) ip op}) \end{aligned} \tag{6.6}$$

Rewriting theorem (6.3) with the theorems (6.5) and (6.6), we obtain a new importing theorem (6.7). This theorem states that the formalization of the MDG results based on the semantics of the MDG formula representation can be imported into the HOL to form a HOL theorem based on the semantics of MDG-HDL.

$$
\begin{array}{ll}
\vdash_{thm} & \forall\ \texttt{IMP SPEC.} \\
& \quad \forall\ \texttt{ip flag op op'.} \\
& \quad\ \texttt{PSEQ ip op op' flag} \\
& \qquad\quad \texttt{(SemProgram\_Formula (TransProgCF (TransProgMC SPEC)))} \\
& \qquad\quad \texttt{(SemProgram\_Formula (TransProgCF (TransProgMC IMP)))} \\
& \qquad\quad \supset (\forall\ \texttt{t. (flag t = T))} \wedge \\
& \forall\ \texttt{ip.}\ \exists\ \texttt{op'. SemProgram SPEC ip op'} \supset \\
& (\forall\ \texttt{ip op. SemProgram IMP ip op} \supset \texttt{SemProgram SPEC ip op)}
\end{array} \tag{6.7}
$$

Therefore, the additional assumption for the design can be proved in terms of the semantics of MDG-HDL

$$\forall\ \texttt{ip.}\ \exists\ \texttt{op'. SemProgram SPEC ip op'} \tag{6.8}$$

The converted theorem which we obtain in HOL is in terms of the semantics of MDG-HDL too.

$$(\forall\ \texttt{ip op. SemProgram IMP ip op} \supset \texttt{SemProgram SPEC ip op)} \tag{6.9}$$

Working with the semantics of a high level language (such as MDG-HDL) makes verification easier and more readable. Combining the importing theorem (5.18) or (5.20) with the translator correctness theorem (3.3) allows our additional assumption to be proved in terms of the semantics of MDG-HDL and the theorem we obtain in HOL to be imported in terms of the semantics of MDG-HDL. Therefore, the low level MDG verification results can be converted into HOL in terms of the semantics of a high level language (MDG-HDL).

In the rest of this section, we give a simple example to illustrate the technical detail about how to formally import the verification results proved in the MDG systems to results about circuits in a form that can be reasoned about in the HOL system.

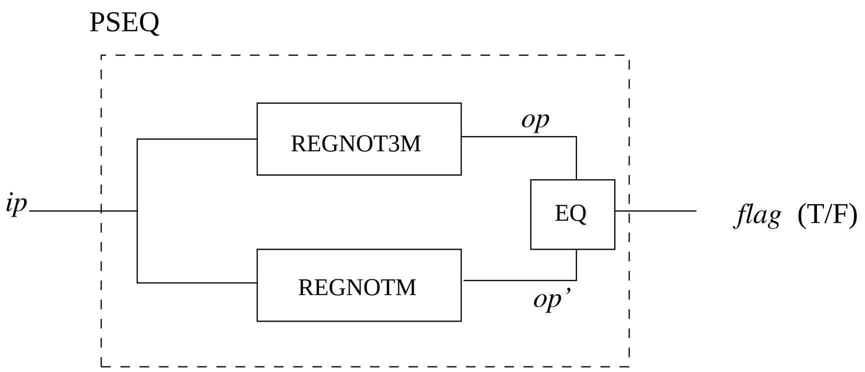


Figure 6.4: The Machine used for Sequential Verification of the REGNOT3M Circuit

Example 2. Consider verifying the sequential circuits in Figure 6.4 using sequential verification. We check that three not gates and a register are equivalent to a single not gate and register. We first prove that the two circuits are equivalent in the MDG system. We next prove the additional assumption in HOL based on the MDG input language – MDG-HDL. Finally, we convert the MDG results into HOL to form the HOL theorem.

Firstly, we prove the circuits using the MDG system. When we use the MDG system to prove the equivalence of these two circuits, we need to specify the circuit description files. The main part of the circuit description file for one NOT gate and one register is

```
...
signal(ip,bool).
signal(op,bool).
signal(x,bool).
component(not_A,not(input(ip),output(x))).
component(reg_A,reg(input(x), output(op))).
init_val(op,0).
outputs([op]).
st_nxst(op,x).
...
```

The main part of the circuit description file for three NOT gates and one register is

```
...
signal(ip,bool).
signal(op,bool).
signal(u,bool).
signal(v,bool).
signal(w,bool).
component(u_comp,not(input(ip),output(u))).
component(v_comp,not(input(u),output(v))).
component(op_comp,not(input(v),output(w))).
component(reg_comp,reg(input(w),output(op))).
outputs([op]).
st_nxst(op,x).
...
```

We also need to provide the algebraic specification file, the symbol order file and the invariant specification file. We input these five files into the MDG system. The MDG verification tool will take the MDG-HDL programs and translate them into two MDG representations. A set of MDG algorithms will be applied to them to obtain their canonical MDG representations. The MDG system will check whether two canonical MDG representations are identical or not and return true or false respectively. In our example, the MDG verification tool returns true so that the two circuits have been successfully proved.

We then define the syntax of the two circuits. The abstract syntax of the first circuit `REGNOT3M` is:

```
IMP = PROG (EXOUT ["op"]) (EXIN ["ip"]) (INV ["u";"v";"w"])
            (SEQ (NOT "ip" "u")
                 (SEQ (NOT "u" "v")
                      (SEQ (NOT "v" "w") (REG "w" "op"))))
```

The abstract syntax of the second circuit REGNOT1M is

```
SPEC = PROG (EXOUT ["op'"]) (EXIN ["ip"]) (INV ["x"])
            (SEQ (NOT "ip" "x") (REG "w" "op"))
```

Since the MDG tool returns ture, we can formalize MDG result into HOL in terms of semantics of the MDG formula representation: the result that MDG proves about PSEQ is that the equality checker is always true. The formalization can be tagged into HOL to form a HOL theorem as shown below:

```
⊢thm     ∀ ip flag op op'.
          PSEQ ip op op' flag
              (SemProgram_Formula (TransProgCF (TransProgMC SPEC)))
              (SemProgram_Formula (TransProgCF (TransProgMC IMP)))
            ⊃ (∀ t.  (flag t = T))                                  (6.10)
```

The next step is to prove the additional assumption based on the semantics of MDG-HDL. Namely, for all possible input traces, the behavior specification REGNOT1 can be satisfied for some output and state traces:

```
⊢thm ∀ ip.  ∃ op'.  SemProgram SPEC ip op'                          (6.11)
```

By instantiating the syntax of the two circuits into the importing theorem for sequential verification (6.7), we obtain a theorem.

```
⊢thm (∀ ip op op' flag.
        PSEQ ip op op' flag
           (SemProgram_Formula (TransProgCF (TransProgMC SPEC)))
           (SemProgram_Formula (TransProgCF (TransProgMC IMP))) ⊃
                  (∀t.  flag t = T)) ∧
     (∀ ip.  ∃ op'.  SemProgram SPEC ip op') ⊃
     (∀ ip  op.  SemProgram IMP ip op ⊃ SemProgram SPEC ip op)
```

Finally, we obtain the conversion theorem by discharging the theorem of formalizing the MDG result (6.10) and the existential theorem (6.11). This theorem states that the implementation implies the specification.

$\vdash_{thm}$ ∀ `ip op. SemProgram IMP ip op` ⊃ `SemProgram SPEC ip op`

6.2 Combining the Translator Correctness Theorem with the Importing Theorems for an Extended Subset

The main idea of this section is similar to that of the last section. However, the syntax and the semantics are different, because we consider an extended subset. Since we only proved the first translator for this subset, the formalization of the MDG result is based on the core MDG-HDL language rather than the MDG formula representation (Figure 6.2).

6.2.1 Combinational Verification

As we mentioned in section 6.1.1, for combinational verification, we only need to formalize MDG verification result and tag it into HOL. The tagged theorem is in the form the HOL system needed. The formalization of MDG verification result based on the semantics of the core MDG-HDL can be given as below:

```
∀ ip op.
        SemProgram_Core (TransProgMC C1) ip op =
                 SemProgram_Core (TransProgMC C2) ip op
```
(6.12)

By using the translator correctness theorem (4.1), we have proved a theorem `Formalize_Eqce_Thm` (6.13) which states that the formalization of the MDG result based on the core MDG-HDL language is equivalent to the formalization of the MDG result based on MDG-HDL.

$$\vdash_{thm} \quad (\forall\ \texttt{ip op.}\ \texttt{SemProgram\_Core (TransProgMC C1) ip op =}\ \texttt{SemProgram\_Core (TransProgMC C2) ip op}) = \forall\ \texttt{ip op.}\ \ \texttt{SemProgram C1 ip op = SemProgram C2 ip op} \tag{6.13}$$

Therefore, the MDG verification results can be converted into HOL to form the HOL theorems in terms of the semantics of MDG-HDL.

6.2.2 Sequential Verification

Similar to the section 6.1.2, we first instantiate the two machines in terms of the semantics of the core MDG-HDL language in the importing theorem (5.18) or (5.20). Therefore, we obtain the importing theorem based on the semantics of the core MDG-HDL language as shown below:

$$\begin{aligned}\vdash_{thm} \quad & \forall\ \texttt{IMP SPEC.} \\ & (\forall\ \texttt{ip op op' flag.} \\ & \quad \texttt{PSEQ ip op op' flag} \\ & \quad\quad \texttt{(SemProgram\_Core (TransProgMC SPEC))} \\ & \quad\quad \texttt{(SemProgram\_Core (TransProgMC IMP))} \\ & \quad\quad\quad \supset\ (\forall\ \texttt{t.}\ \ \texttt{(flag t = T))}) \wedge \\ & (\forall\ \texttt{ip.}\ \ \exists\ \ \texttt{op'.}\ \ \texttt{SemProgram\_Core (TransProgMC SPEC) ip op'}) \supset \\ & (\forall\ \texttt{ip}\ \ \texttt{op.(SemProgram\_Core (TransProgMC IMP) ip op)} \supset \\ & \quad\quad \texttt{(SemProgram\_Core(TransProgMC SPEC) ip op)})\end{aligned} \tag{6.14}$$

Secondly, we need to prove an additional assumption.

$$\vdash_{thm} (\forall\ \texttt{ip.}\ \ \exists\ \ \texttt{op'.}\ \ \texttt{SemProgram\_Core(TransProgMC SPEC) ip op'}) \tag{6.15}$$

By using the translator correctness theorem (4.1), we prove a theorem `Exist_Eqe_Thm` (6.16). This theorem states that the additional assumption based on the semantics of the core MDG-HDL language is equivalent to that based on the semantics of MDG-HDL. In other words, we can prove the additional assumption in terms of the semantics of MDG-HDL.

$$\vdash_{thm} \quad (\forall \texttt{ ip. } \exists \texttt{ op'. (SemProgram\_Core (TransProgMC SPEC))) ip op'}) = \\ (\forall \texttt{ ip. } \exists \texttt{ op'. SemProgram SPEC ip op'}) \tag{6.16}$$

Thirdly, we prove the theorem `Imp_Eqe_Thm`, which states that the tranditional HOL theorem based on the semantics of the core MDG-HDL language is equivalent to that based on the semantics of MDG-HDL.

$$\vdash_{thm} \quad (\forall \texttt{ ip op.} \\ \texttt{(SemProgram\_Core (TransProgMC IMP)) ip op } \supset \\ \texttt{(SemProgram\_Core (TransProgMC SPEC)) ip op}) = \\ (\forall \texttt{ ip op. (SemProgram IMP) ip op } \supset \\ \texttt{(SemProgram SPEC) ip op}) \tag{6.17}$$

Finally, the new importing theorem `Import_Mdghdl_Thm` is obtained by rewriting theorems (6.14) with the theorem (6.16) and (6.17).

$$\vdash_{thm} \quad \forall \texttt{ IMP SPEC.} \\ (\forall \texttt{ ip op op' flag.} \\ \texttt{PSEQ ip op op' flag} \\ \texttt{(SemProgram\_Core (TransProgMC SPEC))} \\ \texttt{(SemProgram\_Core (TransProgMC IMP))} \\ \supset (\forall \texttt{ t. (flag t = T)})) \wedge \\ (\forall \texttt{ ip. } \exists \texttt{ op'. SemProgram SPEC ip op')} \supset \\ (\forall \texttt{ ip op. SemProgram IMP ip op } \supset \\ \texttt{SemProgram SPEC ip op}) \tag{6.18}$$

As a result, combination of the translator correctness theorem and importing theorems allows MDG verification result to be imported into HOL in terms of semantics of MDG-HDL. An example for importing MDG verification result into HOL for the extended subset will be given in Chapter 8.

Summary

We have combined the compiler correctness theorems with the importing theorems based on the deep embedding semantics. This combination allows the MDG results to be reasoned about in HOL in terms of the MDG input language (MDG-HDL). The two different MDG verification applications for two subsets have been formalized in terms of the low level language and imported in a way that corresponds to the semantics of MDG-HDL.

Chapter 7

Existential Theorems

As we stated in Chapter 5, the importing theorem for sequential verification has the form:

$$
\begin{array}{l}
\vdash_{thm} \texttt{Formalized MDG result} \wedge \\
\qquad \forall\ \texttt{ip.}\ \ \exists\ \texttt{op.}\ \ \texttt{SPEC ip op} \supset \\
\qquad\qquad (\forall\ \texttt{ip op.}\ \ (\texttt{IMPL ip op} \supset\ \texttt{SPEC ip op}))
\end{array}
$$

where `SPEC` represents the behavioral specification and `IMPL` represents the structural specification. The first assumption is discharged by the MDG verification. However, for importing the sequential verification results into HOL, a user of the hybrid system strictly needs to prove an additional assumption (an existential theorem) to ensure the correct HOL theorem can be made. This theorem states that for all possible input traces, the behavioral specification `SPEC` can be satisfied for some outputs (i.e., there exists at least one output for which the relation is true):

$$\forall\ \texttt{ip.}\ \ \exists\ \texttt{op.}\ \ \texttt{SPEC ip op} \tag{7.1}$$

When we convert the MDG results into HOL to form the HOL theorems, the theorems actually state that the implementation of the design implements its specification as shown in (7.2).

$$\forall \texttt{ ip op. (IMPL ip op} \supset \texttt{ SPEC ip op)} \tag{7.2}$$

This representation might meet an inconsistent model that trivially satisfies any specification. We need to verify a stronger consistency theorem against the implementation as suggested in [58], which has the form:

$$\forall \texttt{ ip. } \exists \texttt{ op. IMPL ip op} \tag{7.3}$$

This means that for any set of input values `ip` there is a set of output values `op` which is consistent with it. This shows that the model does not satisfy a specification merely because it is inconsistent.

In this chapter, we investigate a way of proving the additional assumption and the stronger consistency theorem based on the syntax and semantics of the MDG input language [82]. As we mentioned above, we prove the additional assumption because we want to make the linking process easier and remove the burden from the user of the hybrid system. We prove the stronger consistency theorem because we want to avoid an inconsistent model occurring. The above two theorems actually have the same form. In the rest of this thesis, we call them `existential theorems`. If we use `C` to represent any specification or implementation of a circuit, ip and `op` to represent the external inputs and outputs, the `existential theorem` should have the form:

$$\forall \texttt{ ip. } \exists \texttt{ op. C ip op} \tag{7.4}$$

For example, if we consider a circuit consisting of two `NOT` gates in series, the existential theorem for this circuit should be:

$$\vdash_{thm} \forall \texttt{ ip. } \exists \texttt{ op. (}\exists \texttt{ op1. SEM\_NOT ip op1} \wedge \texttt{ SEM\_NOT op1 op)}$$

In fact, the stronger consistency theorem (7.3) is an `existential theorem` for the structural specification, whereas the additional assumption (7.1) for the importing theorem is an `existential theorem` for the behavioral specification.

The goal of the `existential theorem` is existentially quantified. We can remove hidden lines in goals of this form using `EXISTS_TAC`, which strips away the leading existentially quantified variable and substitutes `term` for each free occurrence in the body. This `term` is called the `existential term`. An `existential term` of a variable is determined by one or several `output representations` of the corresponding MDG-HDL components. An `output representation` of a component represents an output function of this component, which depends on its input value and output value at the current time or an earlier time instance. There is a HOL tactic, EXISTS_ELIM_TAC [6], which is used to eliminate existentially quantified variables in a goal. This tactic corresponds to a theorem EXISTS_ELIM given below.

$$\vdash_{thm} (\exists\ \mathtt{x}.\ \ (\mathtt{x} = \mathtt{t}) \wedge (\mathtt{A}\ \mathtt{x})) = \mathtt{A}\ \mathtt{t} \tag{7.5}$$

In other words, if the existentially quantified variable (`x`) is explicitly represented by its value as in (7.5) with (x = t) in the goal, the tactic `EXISTS_ELIM_TAC` can be used to remove the hidden lines. The general purpose simplification tactic, `SIMP_TAC` can similarly be used to eliminate existentially quantified variables. However, for dealing with those existentially quantified variables such as (`x`) which are not represented as (x = t), we need to find their `output representations.`

In this chapter, we concentrate on proving the existential theorems based on the syntax and semantics of MDG-HDL [82] [26]. However, a similar method can be used to solve other existentially quantified goals. This is because we provide the `output representation` for each component (mainly logic gates and flip-flops). The `existential term` of a design, which reduces the goal $\exists$ `x.  t` to `t[u/x]`, is determined in terms of the corresponding `output representations`. We also provide tactics for expanding the semantics of the circuit and proving the `existential theorem`.

We have defined semantic functions for two subsets MDG-HDL. For giving a corresponding importing theorem for sequential verification, we need to prove the existential theorem for the implementation of the design in term of the semantics. We need to provide the general `output representation` for each component of the two subsets of the MDG-HDL library. Because the main ideas of defining the `output representation` for each component of the two subsets are same, we will only give the detail about how to define the `output representation` for the extended subset. In other words, we will talk about how to prove the existential theorem for the extended subset.

7.1 Existential Theorem for the Extended Subset

In this section, we provide the general `output representation` for each component in the MDG-HDL library. Because the `existential term` for a design is determined in terms of the `output representation` of its components, these provide a toolkit for then proving the `existential theorem` of the design. We also provide three tactics `EXPAND_SEMANTICS_TAC`, `PROVE_EXIST_TAC` and `PROVE_TABLE_EXIST_TAC` which automatically expand the semantics of the program and prove the goal. The first tactic is used for expanding the semantics of the program (design) and obtaining a goal of the form ∃ `a1 ... an. body`. The tactics `PROVE_EXIST_TAC` and `PROVE_TABLE_EXIST_TAC` are used for verifying goals.

The proof process for proving an existential theorem is divided into three steps. We first expand its semantics and rewrite away the abstract syntax, and obtain the existentially quantified goal. We then strip away the existential quantified variable. Finally, we prove the goal.

Example 1. Consider a circuit that only consists of one `NOT` gate. The abstract syntax of this circuit is represented as:

```
(PROG (EXOUT ["op"])(EXIN ["ip"]) (INV []) (NOT "ip" "op"))
```

The existential theorem for this circuit is

```
⊢thm ∀ ip.  ∃ op.
            SemProgram (PROG (EXOUT ["op"])(EXIN ["ip"])
                   (INV [])(NOT "ip" "op")) ip op
```

Expanding the semantics of the program using the tactic `EXPAND_SEMANTICS_TAC`, we obtain a subgoal which has the form ∃ `a1 ... an. body`. Here:

```
∃ op.  ∀ t.  IS_BOOL (HD ip t) ∧ IS_BOOL (HD op t) ⊃
        ∀t.  MDG_TO_BOOL (HD op t) = ~ MDG_TO_BOOL (HD ip t)
```

The existential theorem of this circuit is existentially quantified by its external output `op`. More detail will be given later.

In the rest of this chapter, we first define the `output representation` for each component in the MDG-HDL library apart from the `TABLE`. We then provide a method to find the `output representation` for the `TABLE` component. We next deal with the existentially quantified internal variable. Finally, we give an example that demonstrates how to apply our approach to prove the `existential theorem` of a whole circuit.

7.2 The Output Representation for the Basic MDG-HDL Components

In the MDG-HDL library, there are two classes of non-table component. In one the output of the component is a signal variable (ie non state holding), in the other the output of the component is a state variable. The `existential terms` for the two classes are slightly different.

(1) The output of a component is signal variable.

Most components in the MDG-HDL library belong to this class having no state component: their output is a signal variable. For stripping away the existentially quantified variable, we have defined the `output representation` for each component. For example, the general `output representation` for the `NOT` gate is defined as

$$\vdash_{def} \texttt{existnot (ip:Mdg\_Basic) =}$$
$$\texttt{(Bool1\_Mdg} \sim \texttt{(}\lambda\texttt{wv. (if wv = BOOL T then T else F)) ip)}$$

where `Bool1_Mdg` is an auxiliary function, which converts a `boolean` value to a `Mdg_Basic` value. This definition states that the function is related to the input `ip`. We use this term as the basis of the witness term for existential quantification elimination (`EXISTS_TAC` in HOL).

In Example 1 above, both external inputs and external outputs are one element lists. The input of the circuit is therefore `(HD ip)` (taking the first element of list `ip`); we therefore use `(HD ip)` to represent our input variable in the existential term rather than `ip`. The output `op` is a `(num->Mdg_Basic) list`. We use `[λ(t:num). existnot (HD ip (t:num))]` to represent the `existential term` of the circuit. It is used to strip away the existentially quantified goal. The second tactic `PROVE_EXIST_TAC` can then be used to prove the goal. The `output representation` for other components in this class can be defined in a very similar way.

(2) The output of a component is a state variable.

In this class, the output value of a component refers to values at an earlier time instance. When we strip away the existentially quantified variable `op`, the time value in the existential term must be one instance earlier.

Example 2. Consider proving an existential theorem for a one register circuit. The `output representation` for a register `existreg` is given below:

```
⊢def existreg (ip:Mdg_Basic) =
        (Bool1_Mdg(λwv.  (if wv = BOOL T then T else F )) ip)
```

We first use the tactic `EXPAND_SEMANTICS_TAC[SEM_REG]` which expands the semantics of the circuit. The existential quantifier elimination tactic `EXISTS_TAC` is then used to strip away the existentially quantified variable `op`. However, the existential term `[(λ(t:num). existreg (HD ip ((t-1):num)))]` is different to the one we described above. Because the output value of the register refers to values at an earlier time instance, the time in function `existreg` is `(t−1)` rather than `t`. Finally, the `existential theorem` for one `register` can be proved by using tactic `PROVE_EXIST_TAC`.

7.3 The Output Representation for TABLE Components

The predefined `TABLE` component must be dealt with separately. There exist three different situations. In each of these situations, the `output representation` of the `TABLE` is based on the output function `existtable` whose definition is given below:

```
⊢def (existtable input [] u_out default t = default t) ∧
     (existtable input vs [] default t = default t) ∧
     (existtable input (CONS v vs) (CONS u u_out) default t =
                        (if (Table_match input v t) then (u t)
                         else (existtable input vs u_out default t)))
```

This definition represents the output value of the table. In the definition, the input of the table `input` is a list. Each element in the list could be used to represent the output value at an earlier time instance. From this definition, we have proved a theorem which states the relation between the predicate `table` and predicate `existtable`. A `table`'s output value at time `t` is equal to the value of predicate `existtable` at time `t`.

```
⊢thm ∀ u_outs u_out t.
     table input op u_outs u_out default t =
          (op t = (existtable input u_outs u_out default t))
```

Now, we will consider how to use `existtable` to give the `output representation` for the three different table situations in turn.

(1) The output of a `TABLE` is a signal variable.

In this situation, the output is a relation of the input and the other three table arguments. The `output representation` for `TABLE` is `existtable ip vs u_out default`. In other words, the function `existtable` represents the output relation.

For example, if we want to prove an existential theorem for the `TABLE` of a `NOT` gate circuit, the `existential term` for the table specifying a `NOT` gate is

```
[existtable [(HD ip) :(num -> Mdg_Basic)]
            [[TABLE_VAL (BOOL T)]; [TABLE_VAL (BOOL F)]]
            [(λt.  BOOL F); (λt.  BOOL T)] (λt.  ARB)]
```

(2) The output of a `TABLE` is a state variable and the input of the `TABLE` does not contain the output variable.

In this case, the output of the `TABLE` at the current time does not depend on itself at an earlier time instance. The existential term refers to the values at an earlier time instance, which is `λt.  existtable ip vs u_out default (t-1)`. The time in function `existtable` is `(t−1)` rather than `t`. For example, if we want to prove an existential theorem for the `TABLE` of a `Register` circuit, the `existential term` which refers to values at an earlier time instance for this circuit is

```
[λt. existtable [(HD ip) :(num -> Mdg_Basic)]
          [[TABLE_VAL (BOOL T)]; [TABLE_VAL (BOOL F)]]
           [(λt. BOOL T); (λt. BOOL F)] (λt. ARB) (t-1)]
```

(3) The output of a `TABLE` is a state variable and the input of the `TABLE` contains the output variable.

In this situation, the output value of the `TABLE` not only depends on inputs but also depends on its own value at an earlier time instance. We cannot give the general `output representation` for this kind of `TABLE`. However, we provide a method through an example to explain how to obtain an `output representation` for the `TABLE`.

Example 3. We consider the following goal for a program containing a table in which the table output value not only depends on inputs but also depends on its own value at an earlier time instance (see Figure 7.1).

After using the tactic `EXPAND_SEMANTICS_TAC` to expand the semantics of the syntax, we obtain:

```
∃ op. (∀ t. (IS_BOOL (HD ip t) ∧ IS_BOOL (HD op t)) ∧
    IS_BOOL ((HD op o NEXT) t)) ⊃
  TABLE [HD (ip :(num -> Mdg_Basic) list); HD op] (HD op (t + 1))
   [[TABLE_VAL (BOOL F); TABLE_VAL (BOOL F)];
    [TABLE_VAL (BOOL F); TABLE_VAL (BOOL T)];
    [TABLE_VAL (BOOL T); TABLE_VAL (BOOL F)];
    [TABLE_VAL (BOOL T); TABLE_VAL (BOOL T)]]
   [(λ(t :num). BOOL F); (λ(t :num). BOOL T); (λ(t :num). BOOL T);
    (λ(t :num). BOOL T)] (λ(t :num). ARB)
```

We notice that the output value at the time `t+1` depends on the output value at the time `t`. For stripping away the existentially quantified variable `op`, we have to

```
∀ ip.
    ∃ op.   SemProgram (PROG (EXOUT ["op"])(EXIN ["ip"]) (INV [])
            (TABLESYN ["ip"; "op"] (NEXTV "op")
                [[TABLE_VAL (BOOL F); TABLE_VAL (BOOL F)];
                 [TABLE_VAL (BOOL F); TABLE_VAL (BOOL T)];
                 [TABLE_VAL (BOOL T); TABLE_VAL (BOOL F)];
                 [TABLE_VAL (BOOL T); TABLE_VAL (BOOL T)]]
                [BOOL F;BOOL T;BOOL T;BOOL T] (DENORMAL ARB))) ip op
```

	INPUT		OUTPUT
	ip t	**op t**	**op (t+1)**
IF	BOOL F	BOOL F	BOOL F
	BOOL F	BOOL T	BOOL T
	BOOL T	BOOL T	BOOL F
	BOOL T	BOOL T	BOOL T
ELSE			ARB

Figure 7.1: The Output of a `TABLE` is a State Variable and Contains in the Input list

define a new constant `existtable_next` of the form:

```
existtable_next ip (SUC t) =
  existtable [HD ip; (λa.  existtable_next ip a)]
     [[TABLE_VAL (BOOL F); TABLE_VAL (BOOL F)];
      [TABLE_VAL (BOOL F); TABLE_VAL (BOOL T)];
      [TABLE_VAL (BOOL T); TABLE_VAL (BOOL F)];
      [TABLE_VAL (BOOL T); TABLE_VAL (BOOL T)]]
     [(λ(t :num).  BOOL F); (λ(t :num).  BOOL T); (λ(t :num).  BOOL T);
      (λ(t :num).  BOOL T)] (λ(t :num).  ARB) t
```

where `((SUC t) = t+1)`. However, we cannot define this function directly in HOL by using the `Define` function because it is not well-defined. In particular, it is of the form

```
f (SUC t) = g f
```

where `f` is `existtable_next` applied to arguments, and `g` is `existtable` applied to arguments. The function is passing `f` (a functional value of type:`num->Mdg_Basic`) to another function. In order to make this valid, we have to show that the functions called by `g` are only called in ways that decrease some measure function. Therefore, we expand the definition first and obtain a well-defined function so as to use `Define` to define this function.

We first expand the definition of the `existtable, Table_match, HD, TL` and `TableVal_to_Val` in order to define `existtable_next` by using `REWRITE_CONV`. We can then obtain a well-defined function and use `Define` to define the function `existtable_next`. We next obtain the existential term which is

```
[((existtable_next (ip:(num->Mdg_Basic) list)) :num->Mdg_Basic)]
```

Finally, the existential goal can be proved by using `PROVE_TABLE_EXIST_TAC`. Therefore, we can prove the existential theorem of the above circuit by using the above three steps as long as we find its `output representations`.

7.4 Dealing with the Existential Quantified Internal Variables

When we prove the existential theorem for a circuit, if the circuit contains internal wires, we also need to strip away these wires. The `existential terms` for these wires are nearly the same as we described above. A difference is that the type of these wires is `:num -> Mdg_Basic` rather than `:(num -> Mdg_Basic) list`. This is because we do not use a list to represent an internal wire.

Example 4. We consider the proof of the `existential theorem` for a circuit consisting of one `AND` gate and one `REGISTER`. The semantics of this circuit is

```
∀ ip.  ∃ op.
     SemProgram (PROG (EXOUT ["op"]) (EXIN ["ip1"; "ip2"]) (INV ["u"])
       (JOIN (AND "ip1" "ip2" "u") (REG "u" "op"))) ip op
```

By expanding the semantics using `EXPAND_SEMANTICS_TAC[SEM_AND, SEM_REG]`, we obtain

```
∃.   x1 op.
   (∀ t.  IS_BOOL (HD ip t) ∧ IS_BOOL (HD (TL ip) t)) ∧
      IS_BOOL (HD op (t + 1))) ⊃
    (∀ t.
       IS_BOOL (x1 t) ∧
       (MDG_TO_BOOL (x1 t) =
       MDG_TO_BOOL (HD ip t) ∧ MDG_TO_BOOL (HD (TL ip) t))) ∧
       IS_BOOL (x1 t) ∧ (MDG_TO_BOOL (HD op (t + 1)) = MDG_TO_BOOL (x1 t))
```

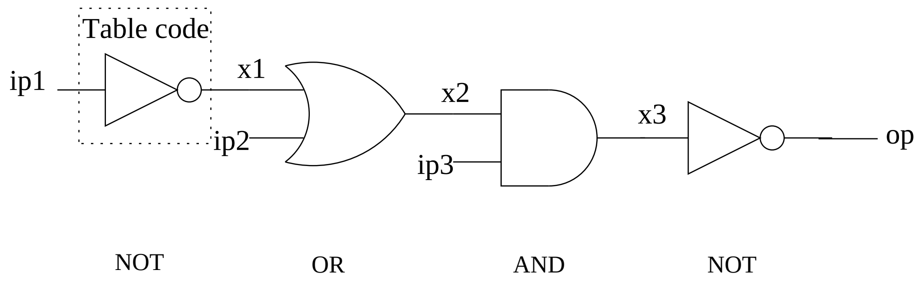


Figure 7.2: A Circuit

where `x1` is an internal wire which is the output of the `AND` gate and the input of the `REGISTER`. It is a `(num -> Mdg_Basic)` term. The existential term of `x1` (`x1_exist`) depends on the `output representation` of the `AND` gate (`existand`).

```
x1_exist = (λ(t:num). existand (HD ip (t:num)) (HD (TL ip) t))
```

`op` represents an external output, it is a `(num -> Mdg_Basic) list` term. The output of the `REGISTER` is the only element of this list. Thus the corresponding existential term depends on the `output representation` of the `REGISTER`.

```
[(λ(t:num). (existreg (x1_exist (t-1))))]
```

The tactic `EXISTS_TAC` can then be used to strip away the existentially quantified external variable `op` and internal variable `x1`. Finally, the theorem can be proved by using tactic `PROVE_EXIST_TAC`.

7.5 An Example

Example 5. Consider the circuit shown in Figure 7.2. We will prove the existential theorem of this circuit to illustrate how our approach is deployed with a circuit containing a combination of the situations considered: internal wires, a table, a register and combinational components. The existential theorem for this circuit is represented as:

```
⊢thm ∀ ip.
      ∃ op.
        SemProgram(PROG (EXOUT ["op1"]) (EXIN ["ip1"; "ip2"; "ip3"])
                     (INV ["x1"; "x2"; "x3])
         (JOIN (TABLESYN ["ip"] (NOWV "op")
                 [[TABLE_VAL (BOOL T)];[TABLE_VAL (BOOL F)]]
                 [BOOL F; BOOL T] (DENORMAL ARB))
               (JOIN (OR "x1" "ip2" "x2")
                     (JOIN (AND "x2" "ip3" "x3")
                           (NOT "x3" "op1")))) ip op
```

The proof process can be divided into three steps. We first use the tactic EXPAND_SEMANTICS_TAC to expand the semantics of the syntax. We obtain:

```
∃ x1 x2 x3 op.
 (∀t.  IS_BOOL (HD ip t) ∧ IS_BOOL (HD (TL ip) t) ∧
    IS_BOOL (HD (TL (TL ip)) t) ∧ IS_BOOL (HD op t)) ⊃
    TABLE [HD ip] x1 [[TABLE_VAL (BOOL T)]; [TABLE_VAL (BOOL F)]]
    [(λt.  BOOL F); (λt.  BOOL T)] (λt.  ARB) ∧
 (∀t.  (IS_BOOL (x1 t) ∧ IS_BOOL (x2 t) ∧
     (MDG_TO_BOOL (x2 t) =
        MDG_TO_BOOL (x1 t) ∨ MDG_TO_BOOL (HD (TL ip) t))) ∧
     (IS_BOOL (x2 t) ∧ IS_BOOL (x3 t) ∧
     (MDG_TO_BOOL (x3 t) =
       MDG_TO_BOOL (x2 t) ∧ MDG_TO_BOOL (HD (TL (TL ip)) t))) ∧
     IS_BOOL (x3 t) ∧ (MDG_TO_BOOL (HD op t) = ∼ MDG_TO_BOOL (x3 t)))
```

where x1, x2, x3 are internal wires, op is an external wire list which is one element list [op1]. ip is an external input list, which contains three elements [ip1; ip2; ip3].

We then strip away the existential quantified goal. The internal variable `x1` is the output of the `NOT` gate (TABLE representation) and the input of the `OR` gate. The `output representation` for stripping away this variable is determined by the `NOT` TABLE, which is represented as `x1_exist`.

```
x1_exist = existtable [(HD ip)] [[TABLE_VAL (BOOL T)];
                       [TABLE_VAL (BOOL F)]]
                       [(λt.  BOOL F); (λt.  BOOL T)] (λt.  ARB)
```

The internal variable `x2` is the output of the `OR` gate and the input of the `AND` gate. The `existential term` is determined by the `output representation` of the `OR` gate, which is represented as `x2_exist`.

```
x2_exist = (λ (t:num).   existor (x1_exist t) (HD (TL ip) t))
```

where `x1_exist` is the input of the `OR` gate. The `output representation` is in terms of its input. Similarly, the internal variable `x3` is the output of the `AND` gate and the input of the `NOT` gate. The `existential term` is determined by the `output representation` of the `AND` gate, which is represented as `x3_exist`.

```
x3_exist = (λ (t:num).   existand (x2_exist t) (HD (TL (TL ip)) t))
```

Finally, the external output is the output of a `NOT` gate; the `existential term` is determined by `output representation` of the `NOT` gate.

```
op_exist = (λ (t:num).   existnot (x3_exist t)
```

After stripping away the existentially quantified variables using the above terms, we can finally prove the goal using tactic `PROVE_EXIST_TAC`.

This example demonstrates that knowing the `output representation` for each component in the MDG-HDL component library is practically useful when finding a proper `existential term` of the whole circuit. For any circuit in MDG-HDL, as long as we find the corresponding `existential term` of the circuit, the `existential theorem` of this circuit can be proved.

Although we concentrate on proving the existential theorem for the specification and implementation of a design based on the syntax and semantics of MDG-HDL in this thesis, our methods can be used to solve other HOL goals which are existentially quantified. In fact, we have developed a library for giving the `output representation` of each component in a boolean subset. It can be used to construct the `existential term`, which strips away the existentially quantified variable in the HOL goal. In other words, our `existential terms` and `output representations` can be used to solve some existential quantified HOL goal in other applications.

Summary

In this chapter, we investigate existential theorems based on the syntax and semantics of the MDG input language (MDG-HDL) in HOL. We define an output representation for each component in the MDG-HDL component library. We summarize a general method which is used to prove the existential theorem for any MDG-HDL program. The method can also be used to solve other existentially quantified goals.

Chapter 8

Case Study: Verification of the Correctness and Usability Theorems of a Vending Machine

Up to now, we have proved some translator correctness theorems and some importing theorems. We have combined the translator correctness theorems with the importing theorems. The combination allows the MDG verification results to be imported into HOL in terms of the semantics of MDG-HDL. However, how can we ensure this method is feasible in practice? In other words, how can we ensure the low level MDG verification results can be imported into HOL to form the traditional HOL theorems? Moreover, can the importing theorems be used in HOL?

In this chapter, we will use a simple example, the verification of a correctness theorem and a usability theorem of a vending machine (chocolate machine), to answer the above questions. This example was originally used to verify the absence of post-completion errors within the framework of a traditional hardware verification by Curzon and Blandford [25] [24]. In this work, the correctness of the vending machine was verified, ie it was proved that the implementation of the vending machine meets its specification. A usability property based on its `specification` was then proved. By combining the above two theorems, the usability theorem based on its `implementation` was proved. All the formalization and verification were implemented in HOL.

In our case study, we follow their steps. However, we use the MDG system to verify the correctness of the chocolate machine and formally import the MDG verification result into HOL to form the HOL theorem based on the deep embedding semantics of the MDG input language (MDG-HDL). We then prove the `specification` based usability theorem in the HOL system. By combining those two theorems, one the correctness theorem of the chocolate machine which is verified in MDG (the importing theorem), the other the `specification` based usability theorem which is proved in HOL, we obtain the `implementation` based usability theorem. Therefore, the importing theorem (the correctness theorem) can not only be imported into HOL but also can be used in HOL.

When we use the MDG system to verify the chocolate machine, we give a hardware implementation of the machine and verify it against the specification of a finite state machine. Both are described in the MDG input language (MDG-HDL) and verified in the MDG system. After we verify the correctness of the chocolate machine in the MDG system, the theorem about the formalization of the MDG verification result can be tagged into HOL in terms of the syntax and semantics of the core MDG-HDL language. The importing theorem for the chocolate machine can be obtained by instatiating the theorem (6.18) with the syntax (MDG-HDL) of the implementation and specification of the chocolate machine. We also prove the existential theorem based on semantics of MDG-HDL for the implementation of the chocolate machine using the method we proposed in Chapter 7. Finally, a correctness theorem based on the semantics of MDG-HDL of the chocolate machine, which states that the specification implies the implementation, is obtained.

When we prove the usability theorem based on its `specification` in HOL, we follow the idea of Curzon & Blandford [24]. However, the specification of the chocolate machine is different to theirs. This is because the specification in MDG must be in the form of a finite state machine or table description. Another difference is that we have to add some reasonable assumptions to cope with the different sorts of inputs of the `TABLE`. By combining the correctness theorem and the `specification` based usability theorem, we can obtain the `implementation` based usability theorem. More detail will be discussed in section 8.3.

During this case study, we will show the detail about how to define the syntax and the semantics of the specification and implementation, how to use a new type `Mdg_Basic` to accommodate the different sorts of the inputs for the `TABLE`, how to prove the existential theorem and how to formally import the MDG verification results into HOL to form the HOL theorems and make use of the theorems. We will also explain why the assumptions of the usability theorem are reasonable. In other words, we will go though the methods proposed in Chapter 4, Chapter 5, Chapter 6 and Chapter 7. We will use this example to prove the feasibility of the methodology of our research. This is very important. Since we formally import the MDG verification results into HOL on the trusted MDG system, the degree of trust of the linkage between the MDG and HOL system is high. If our methodology is feasible, it can be used in developing a hybrid system. This will greatly increase the trustworthiness of the hybrid system.

In the rest of this chapter, we will first briefly introduce the chocolate machine in section 8.1. We then verify the machine using the MDG system in section 8.2. We next consider the importation process which formally imports the MDG verification results into HOL to form the HOL theorems in section 8.3. In section 8.4, we prove the `specification` based usability theorem in HOL and prove the `implementation` based usability theorem by making use of the above two proved theorems.

8.1 Chocolate Machine

The chocolate machine is used to sell chocolate as shown in Figure 8.1. It takes pound coins only, returning 20p change. To get the change a button must be pressed. Similarly a further button must be pressed to get the chocolate. The machine has lights next to the coin slot and 2 buttons to indicate the order things should be done. The lights light up to indicate the next action the user should perform. The order of operation is that a coin is inserted, the change button is pressed and the change removed, and then finally the chocolate button is pressed and the chocolate removed. If the user does not press the appropriate button the machine does nothing until the correct button is pressed.

The chocolate machine has three inputs which correspond to the buttons being pressed and a coin inserted. It has five outputs which correspond to three lights and a signal each to release change and chocolate.

8.2 Proving the Chocolate Machine using the MDG System

In this section, we will use the sequential verification of the MDG system to prove the correctness of the chocolate machine. For sequential verification, we need to provide five kinds of input files: the circuit description files (the implementation file and the specification file), the algebraic specification file, the symbol order file and the invariant specification file. The implementation file and the specification file have the same inputs (InsertCoin, PushChange, PushChoc) but different outputs. We use (CoinLight_a, ChocLight_a, ChangeLight_a, GivenChoc_a, GivenChange_a) to represent the outputs in the implementation file and (CoinLight, ChocLight, ChangeLight, GiveChoc, GiveChange) to represent the outputs in the specification file. We will explain the four different files in turn in the following subsections.

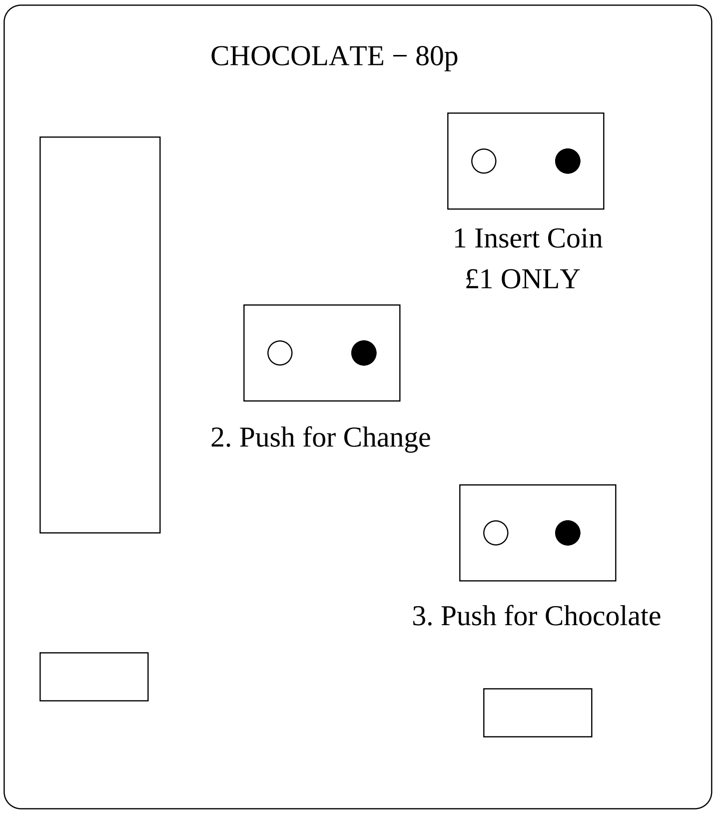


Figure 8.1: The Chocolate Machine

8.2.1 The Implementation

The chocolate machine is implemented in hardware as shown in Figure 8.2. We can use the predefined components in the MDG-HDL library to represent the corresponding circuit as described in [25]. In the circuit, two registers (X and Y) are needed to store the 4 internal states of the chocolate machine (reset, coin, choc, change). The inputs are connected to wire `xin` and `yin` and their outputs to wires `x` and `y`, respectively. In MDG-HDL, we use command `component` to specify their specifications.

```
component(reg_x,reg(input(xin),output(x))).
component(reg_y,reg(input(yin),output(y))).
```

The following representation of abstract states is used:

```
        X  Y
reset   0  0
coin    0  1
change  1  1
choc    1  0
```

The output side of the circuit involves using `NOT` gate and `AND` gate to turn the `x` and the `y` values into 4 signals representing these states.

```
component(out_inv_x,not(input(x),output(xbar))).
component(out_inv_y,not(input(y),output(ybar))).
component(out_and_xy, and(input(x,y),output(change))).
component(out_and_xybar, and(input(x,ybar),output(choc))).
component(out_and_xbary, and(input(xbar,y),output(coin))).
component(out_and_xbarybar, and(input(xbar,ybar),output(reset))).
```

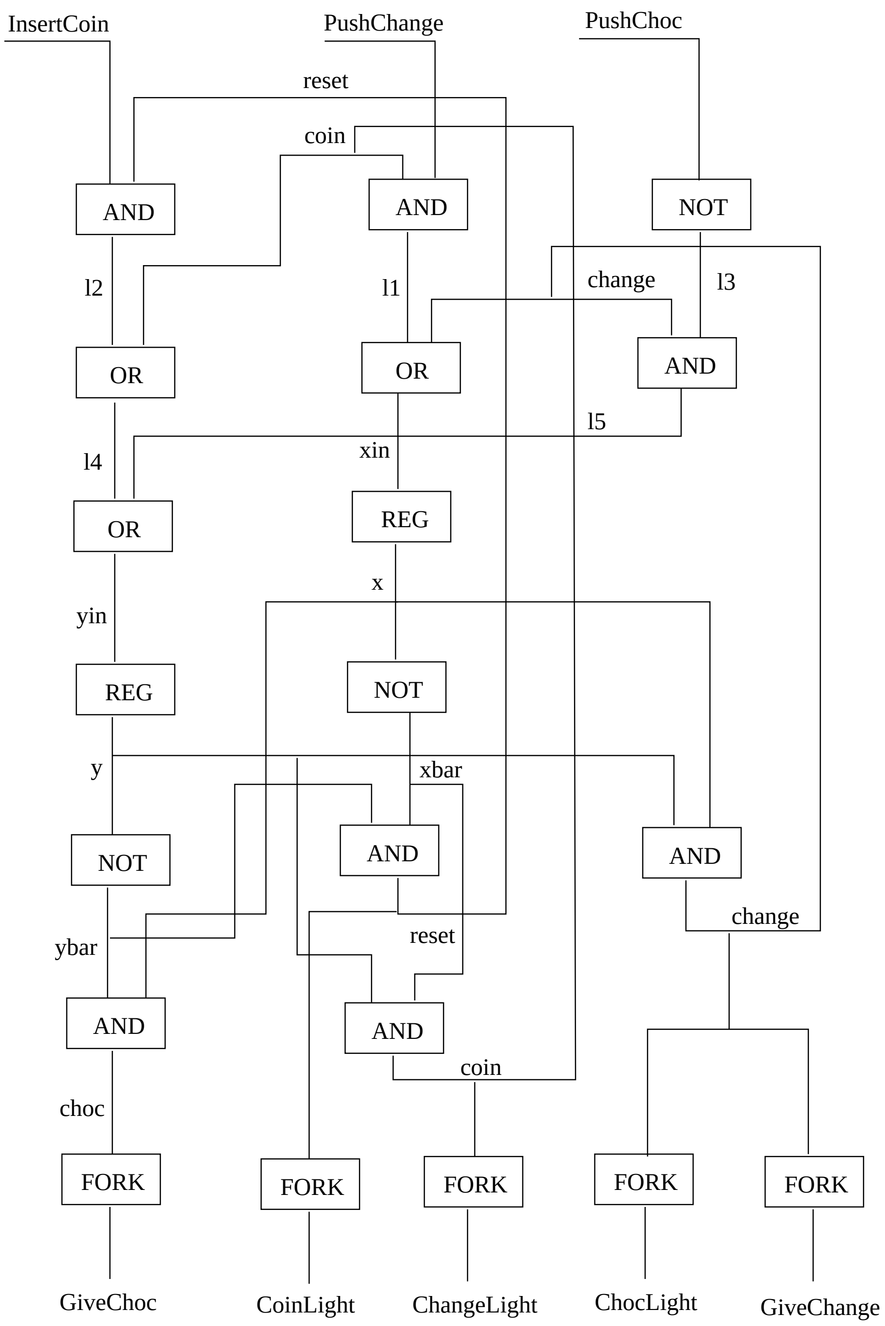


Figure 8.2: The Circuit of the Chocolate Machine

These signals are then wired to the appropriate outputs. The coin light is wired to the `reset` signal, the change light to the `coin` signal, the chocolate light and the mechanism to release the change to the `change` signal.

```
component(wire_choc_givenchoc,fork(input(choc),output(GivenChoc_a))).
component(wire_choc_changlight,fork(input(change),output(ChocLight_a))).
component(wire_change_givechange,fork(input(change),output(GivenChange_a))).
component(wire_coin_choclight,fork(input(coin),output(ChangeLight_a))).
component(wire_reset_coinlight,fork(input(reset),output(CoinLight_a))).
```

The input side of the circuit combines the inputs with the signals representing the states. Signal `x` is `1` in the next state if

(1) we are in the `coin` state `AND` the change button is pressed `OR`

(2) we are in the `change` state.

This is given as:

```
component(x_and,and(input(coin,PushChange),output(l1))).
component(x_or,or(input(change,l1),output(xin))).
```

Signal `y` is `1` in the next state if

(1) we are in the `coin` state `OR`

(2) we are in the `change` state `AND` the chocolate button is `NOT` pressed `OR`

(3) we are in the `reset` state `AND` a coin is inserted.

```
component(y_and_rein, and(input(reset,InsertCoin), output(l2))).
component(y_or_col2, or(input(coin,l2),output(l4))).
```

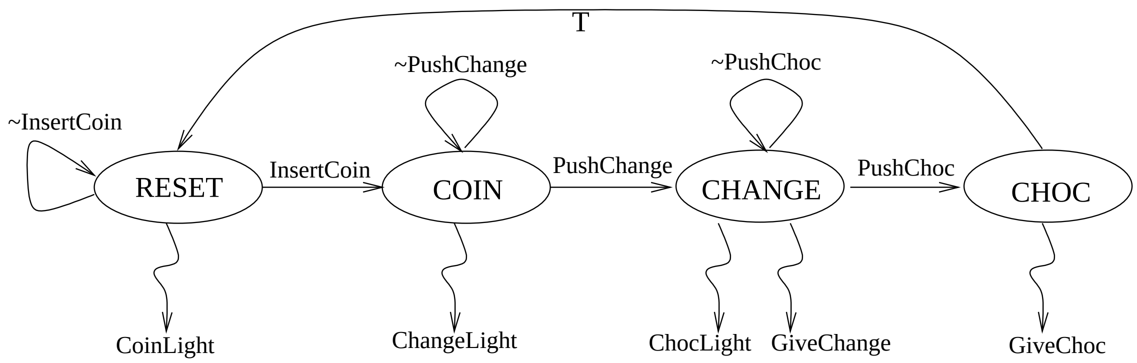


Figure 8.3: The State Transition Diagram of the Chocolate Machine

```
component(y_inv, not(input(PushChoc),output(l3))).
component(y_and_chl3, and(input(change,l3),output(l5))).
component(y_or_l4l5, or(input(l4,l5), output(yin))).
```

We thus obtain the hardware implementation.

8.2.2 The Specification

The MDG specification description is given by a tabular representation of the transition/output relation `TABLE`. We formally specify the chocolate machine as a finite state machine with 4 states – (`RESET, COIN, CHANGE, CHOC`) (see Figure 8.3).

The `RESET` state is the initial state. Each of the other states represent the corresponding action having been done: in the `COIN` state a coin has been accepted; in the `CHOC` state the chocolate is dispensed and in the `CHANGE` state the change is dispensed.

We first define a table which specifies the relations among the current state, inputs and next state. If the machine is in the `RESET` state with the insert coin light lit, the next state is `COIN`. If the machine is in the `COIN` state without the insert light lit, the next state is `RESET`. If the machine is in the `COIN` state with the push change light lit, the next state is `CHANGE`. If the machine is in the `COIN` state without the push change light lit, the next state is `COIN`. If the machine is in the `CHANGE` state with the push chocolate light lit, the next state is `CHOC`. Otherwise the next state is `RESET`. The " $*$ " is used to represent `don't care`

```
component(choc_machine,
  table([[ChocSt,InsertCoin,PushChange,PushChoc, n_ChocSt],
         [RESET,1,*,*,COIN],[RESET,0,*,*,RESET],
         [COIN,*,1,*,CHANGE],[COIN,*, 0, *, COIN],
         [CHANGE,*,*,1,CHOC],[CHANGE,*,*,0,CHANGE],
         [CHOC,*,*,*,RESET]])).
```

For each state we define a table to represent the relation between the states and the outputs. If the machine is in the `RESET` state then the coin light should be on, otherwise the coin light should be off.

```
component(coin_light, table([[ChocSt,CoinLight], [RESET, 1] | 0])).
```

If the machine is in the `COIN` state then the change light should be on, otherwise the change light should be off.

```
component(change_light, table([[ChocSt,ChangeLight], [COIN, 1] | 0])).
```

If the machine is in the `CHANGE` state, the chocolate light should be on and the change should be given. Otherwise, the chocolate light should be off and the change should not be given.

```
component(give_change, table([[ChocSt,GiveChange], [CHANGE, 1] | 0])).
component(choc_light, table([[ChocSt,ChocLight], [CHANGE, 1] | 0])).
```

If the machine is in the `CHOC` state then the chocolate should be given, otherwise the chocolate should not be given.

```
component(give_choc, table([[ChocSt,GiveChoc], [CHOC, 1] | 0])).
```

8.2.3 Three Other Specification Files

We have provided the specification file and the implementation file of the chocolate machine. We also need to provide the algebraic specification file, the symbol order file and the invariant specification file. The algebraic specification file declares sorts, function types and generic constants.The algebraic specification file of the chocolate machine specifies the new concrete sort `ChocStates` which has four different states.

```
conc_sort(ChocStates, [RESET, COIN, CHOC, CHANGE]).
```

The symbol order file provides the custom (user-defined) symbol order for all the variables and cross-operators which would be used in the MDG algorithms. The invariant specification file specifies the invariant condition to be checked during reachability analysis. The full MDG-HDL programs are given in Appendix C.

We input these five files into the MDG system. The MDG verification tool begins to check whether the outputs of the specification file are identical to those of the implementation file or not and returns true or false respectively. In our verification, the MDG system returns true. In other words, the correctness of the chocolate machine has been successfully proved by using the MDG system.

8.3 The Importation Process of the Verification Results

In the last section, the chocolate machine was verified by using the MDG system. In this section, we will show how to import the MDG result into HOL to form the HOL theorems. As we described in Chapter 6, the MDG verification result can be formalized and tagged into HOL in terms of the semantics of the core MDG-HDL. In order to do so, we need to define the syntax and semantics of the specification and implementation of the chocolate machine in HOL. We make use of the importing theorem for sequential verification (6.18) and prove the existential theorem for the implementation of the chocolate machine. The correctness theorem in the traditional HOL form can be obtained. This theorem states that the implementation implies the specification.

8.3.1 The Syntax and the Semantics of the Chocolate Machine

The abstract syntax of MDG-HDL for the specification and implementation of the chocolate machine can be given as we mentioned in Chapter 4 in terms of the MDG input files – the algebraic specification file, the specification file and the implementation file. As we mentioned before, the algebraic specification file declares sorts, function types and generic constants used in the hardware description. When we define the abstract syntax for the specification and implementation files, this part of information should be provided in the declaration of the specification and implementation files respectively. However, since we only consider declaring a sequence of concrete sorts at present, there is no need to declare it in the declaration. We can use any string to represent one concrete sort as we discussed for the extended subset.

The abstract syntax of the MDG-HDL program consists of an external output string list, an external input string list, an internal string list and a `component term`. In both the specification and implementation files of the chocolate machine, we use a three element list `["InsertCoin"; "PushChange"; "PushChoc"]` to represent the abstract syntax of the external inputs and a five element list `["CoinLight"; "ChocLight"; "ChangeLight"; "GiveChoc"; "GiveChange"]` to represent the external outputs. The internal wires list and the `component term` of both files are different as described below.

In the specification file, a one element list `ChocSt` is used to represent the internal variable, whose value could be one of the four states. Its `component term` consists of six `TABLESYN` constructors that are composed by constructor `JOIN`. The full syntax of the specification file of the chocolate machine is given in Figure 8.4. For convenience, in the rest of this section we will use `Choc_Spe_Syn` to informally represent the abstract syntax of the specification.

In the implementation file, there are 15 internal variables. They are represented by a string list `["l1"; "l2"; "l3"; "l4"; "l5"; "xin"; "yin"; "x"; "y"; "xbar"; "ybar"; "choc"; "change"; "coin"; "reset"]`. The `component term` consists of some basic logic gates (`AND, NOT, OR` gate), `FORK` and `REGISTER` which are composed by constructor `JOIN`. The full syntax of the implementation file of the chocolate machine is given in Figure 8.5. In the rest of this section we will use `Choc_Imp_Syn` to informally represent the syntax of the implementation

As we mentioned in Chapter 3 and 4, the semantics of any circuit is described by `SemProgram`, which explicitly represents the relation between the external inputs and the external outputs. In the semantic function, we use a list `ip` to represent external inputs and a list `op` to represent external outputs. In this case, all the formalizations can be represented explicitly with the external inputs `ip` and outputs `op`. The semantics of the specification and implementation files are given below:

$\vdash_{def}$ ∀ `ip op.  CHOC_MACHINE_SPEC ip op = SemProgram Choc_Spe_Syn ip op`
$\vdash_{def}$ ∀ `ip op.  CHOC_MACHINE_IMPL ip op = SemProgram Choc_Imp_Syn ip op`

By expanding the semantics of the program in HOL, we obtain the specification and implementation of the chocolate machine which represent the relation between the external inputs and external outputs.

As we mentioned in Chapter 4, when we define the semantics of the program

```
(PROG
 (EXOUT ["CoinLight"; "ChocLight"; "ChangeLight"; "GiveChoc"; "GiveChange"])
 (EXIN ["InsertCoin"; "PushChange"; "PushChoc"] )
 (INV ["ChocSt"])
 (JOIN (TABLESYN ["ChocSt"; "InsertCoin"; "PushChange"; "PushChoc"]
         (NEXTV( "ChocSt"))
         [[TABLE_VAL (CONCRETE "RESET"); TABLE_VAL (BOOL T); DONT_CARE; DONT_CARE];
         [TABLE_VAL (CONCRETE "RESET"); TABLE_VAL (BOOL F); DONT_CARE; DONT_CARE];
         [TABLE_VAL (CONCRETE "COIN"); DONT_CARE; TABLE_VAL (BOOL T); DONT_CARE];
         [TABLE_VAL (CONCRETE "COIN"); DONT_CARE; TABLE_VAL (BOOL F); DONT_CARE];
         [TABLE_VAL (CONCRETE "CHANGE"); DONT_CARE; DONT_CARE; TABLE_VAL (BOOL T)];
         [TABLE_VAL (CONCRETE "CHANGE"); DONT_CARE; DONT_CARE; TABLE_VAL (BOOL F)];
         [TABLE_VAL (CONCRETE "CHOC"); DONT_CARE; DONT_CARE; DONT_CARE]]
         [(CONCRETE "COIN"); (CONCRETE "RESET"); (CONCRETE "CHANGE");
         (CONCRETE "COIN"); (CONCRETE "CHOC"); (CONCRETE "CHANGE");
         (CONCRETE "RESET")]
          (DENORMAL (CONCRETE "RESET")))
 (JOIN (TABLESYN ["ChocSt"] (NOWV ("CoinLight"))
         [[TABLE_VAL (CONCRETE "RESET")]] [BOOL T] (DENORMAL (BOOL F)))
 (JOIN (TABLESYN ["ChocSt"] (NOWV ("ChangeLight"))
         [[TABLE_VAL (CONCRETE "COIN")]] [BOOL T] (DENORMAL (BOOL F)))
 (JOIN (TABLESYN ["ChocSt"] (NOWV ("GiveChange"))
         [[TABLE_VAL (CONCRETE "CHANGE")]] [BOOL T] (DENORMAL (BOOL F)))
 (JOIN (TABLESYN ["ChocSt"] (NOWV ("ChocLight"))
         [[TABLE_VAL (CONCRETE "CHANGE")]] [BOOL T] (DENORMAL (BOOL F)))
      (TABLESYN ["ChocSt"] (NOWV ("GiveChoc")) [[TABLE_VAL (CONCRETE "CHOC")]]
         [[TABLE_VAL (CONCRETE "CHOC")]] [BOOL T] (DENORMAL (BOOL F)))))))))
```

Figure 8.4: The Abstract Syntax of the Specification File

```
(PROG (EXOUT ["CoinLight"; "ChocLight"; "ChangeLight"; "GiveChoc"; "GiveChange"])
      (EXIN ["InsertCoin"; "PushChange"; "PushChoc"] )
      (INV ["l1"; "l2"; "l3"; "l4"; "l5"; "xin"; "yin"; "x"; "y";
             "xbar"; "ybar"; "choc"; "change"; "coin"; "reset"])
      (JOIN (AND "coin" "PushChange" "l1")
      (JOIN (OR "change" "l1" "xin")
      (JOIN (AND "reset" "InsertCoin" "l2")
      (JOIN (OR "coin" "l2" "l4")
      (JOIN (NOT "PushChoc" "l3")
      (JOIN (AND "change" "l3" "l5")
      (JOIN (OR "l4" "l5" "yin")
      (JOIN (REG "xin" "x")
      (JOIN (REG "yin" "y")
      (JOIN (NOT "x" "xbar")
      (JOIN (NOT "y" "ybar")
      (JOIN (AND "x" "y" "change")
      (JOIN (AND "x" "ybar" "choc")
      (JOIN (AND "xbar" "y" "coin")
      (JOIN (AND "xbar" "ybar" "reset")
      (JOIN (FORK "choc" "GiveChoc")
      (JOIN (FORK "change" "ChocLight" )
      (JOIN (FORK "change" "GiveChange" )
      (JOIN (FORK "coin" "ChangeLight" )
            (FORK "reset" "CoinLight" ))))))))))))))))))))
```

Figure 8.5: The Abstract Syntax of the Implementation File

for the extended subset, we have to add assumptions so as to avoid the sort of each variable being mismatched and inconsistent model being produced. The assumptions are to make sure each of the external inputs and outputs has proper sort (either (`BOOL bool`) terms or (`CONCRETE string`) terms). For example, the semantics of the specification of the chocolate machine (Figure 8.6) states that if the external inputs and outputs are boolean values then the semantics of the program will be six `TABLEs` connected together. In Figure 8.6, one of the inputs of the first `TABLE` is `ChocSt`. The value `ChocSt` can only be one of the four states, but the value of the external inputs can only be a boolean value. The new type `Mdg_Basic` is defined to deal with this situation. Similarly, the implementation of the chocolate machine can be obtained.

8.3.2 Importing the MDG Results into HOL

As we stated in Chapter 6, the importing theorem for the chocolate machine can be obtained by instantiating theorem (6.18) with the syntax of its implementation and specification (`Choc_Spe_Syn` and `Choc_Imp_Syn`).

```
val Import_Choc_Thm =
        (SPECL[--`Choc_Spe_Syn`--, --` Choc_Imp_Syn `--]Import_Mdghdl_Thm);
```

We obtain the theorem `Import_Choc_Thm`

```
⊢thm   (∀ ip flag op op'.
            PSEQ ip flag op op'
                (SemProgram_Core (TransProgMC Choc_Imp_Syn))
                (SemProgram_Core (TransProgMC Choc_Spe_Syn))
                         ⊃ (∀ t.  (flag t = T)) ∧
          ∀ ip.  ∃  op'.  SemProgram Choc_Spe_Syn ip op' ⊃
          (∀ ip op.  SemProgram Choc_Imp_Syn ip op ⊃
                           SemProgram Choc_Spe_Syn ip op)          (8.1)
```

```
(∀ t.  IS_BOOL (HD ip t) ∧ IS_BOOL (HD (TL ip) t) ∧
      IS_BOOL (HD (TL (TL ip)) t) ∧ IS_BOOL (HD op t) ∧
      IS_BOOL (HD (TL op) t) ∧ IS_BOOL (HD (TL (TL op)) t) ∧
      IS_BOOL (HD (TL (TL (TL op))) t) ∧
      IS_BOOL (HD (TL (TL (TL (TL op)))) t)) ⊃
(∃ ChocSt.
    (TABLE [ChocSt; (HD ip); (HD (TL ip)); (HD (TL(TL ip)))] (( ChocSt) o NEXT)
           [[TABLE_VAL (CONCRETE "RESET"); TABLE_VAL (BOOL T); DONT_CARE; DONT_CARE];
           [TABLE_VAL (CONCRETE "RESET"); TABLE_VAL (BOOL F); DONT_CARE; DONT_CARE];
           [TABLE_VAL (CONCRETE "COIN"); DONT_CARE; TABLE_VAL (BOOL T); DONT_CARE];
           [TABLE_VAL (CONCRETE "COIN"); DONT_CARE; TABLE_VAL (BOOL F); DONT_CARE];
           [TABLE_VAL (CONCRETE "CHANGE"); DONT_CARE; DONT_CARE; TABLE_VAL (BOOL T)];
           [TABLE_VAL (CONCRETE "CHANGE"); DONT_CARE; DONT_CARE; TABLE_VAL (BOOL F)];
           [TABLE_VAL (CONCRETE "CHOC"); DONT_CARE; DONT_CARE; DONT_CARE]]
            [(λt.  CONCRETE "COIN"); (λt.  CONCRETE "RESET");
            (λt.  CONCRETE "CHANGE"); (λt.  CONCRETE "COIN");
            (λt.  CONCRETE "CHOC"); (λt.  CONCRETE "CHANGE");
            (λt.  CONCRETE "RESET")] (λt.  (CONCRETE "RESET")) t) ∧
    (TABLE [ChocSt] (HD op) [[TABLE_VAL (CONCRETE "RESET")]] [TSIG1] (FSIG1)) ∧
    (TABLE [ChocSt] (HD(TL(TL op))) [[TABLE_VAL (CONCRETE "COIN")]]
               [TSIG1] (FSIG1)) ∧
    (TABLE [ChocSt] (HD(TL(TL(TL(TL op))))) [[TABLE_VAL (CONCRETE "CHANGE")]]
                [TSIG1] (FSIG1)) ∧
    (TABLE [ChocSt] (HD (TL op)) [[TABLE_VAL (CONCRETE "CHANGE")]]
                [TSIG1] (FSIG1)) ∧
    (TABLE [ChocSt] (HD(TL(TL(TL op)))) [[TABLE_VAL (CONCRETE "CHOC")]]
                [(TSIG1)] (FSIG1)
```

Figure 8.6: The Semantics of the Specification File

Since the MDG tool have verified the correctness of the chocolate machine, the theorem about the formalization of the MDG verification result can be tagged into HOL in terms of the semantics of core MDG-HDL.

$$
\begin{array}{ll}
\vdash_{thm} & (\forall\ \texttt{ip flag op op'}. \\
& \quad \texttt{PSEQ ip flag op op'} \\
& \qquad \texttt{(SemProgram\_Core (TransProgMC Choc\_Imp\_Syn))} \\
& \qquad \texttt{(SemProgram\_Core (TransProgMC Choc\_Spe\_Syn))} \\
& \qquad\qquad \supset (\forall\ \texttt{t.}\ \ \texttt{(flag t = T)}))
\end{array}
\tag{8.2}
$$

We then prove the additional assumption by using the method we proposed in Chapter 7. This theorem states that for all possible input traces, the behavior specification `(SemProgram Choc_Spe_Syn ip op')` can be satisfied for some output and state traces (i.e., there exists at least one output and state trace for which the relation is true):

$$\forall\ \texttt{ip.}\ \ \exists\ \texttt{op'.}\ \ \texttt{(SemProgram Choc\_Spe\_Syn ip op')} \tag{8.3}$$

After expanding the semantics by using `EXPAND_SEMANTICS_TAC []`, we obtain a subgoal as shown in Figure 8.7. It is existentially quantified by two variables `x1, op`. Variable `x1` is an internal wire variable with type `:(num -> Mdg_Basic)`, but variable `op` is an external output with type `:((num -> Mdg_Basic) list)`.

Firstly, we need to find the `existential term` for internal variable `x1`. The variable `x1` is a state variable, it is an output of a `TABLE` and the input of the other `TABLE`s. As we mentioned in section 7.2, the output value of the `TABLE` not only depends on inputs but also depends on its own value at an earlier time instance. In this situation, the `existential term` for the variable `x1` can be obtained as we introduced in Chapter 7. We use `REWRITE_CONV` to expand the semantics of `existtable, Table_match, HD, TL, TableVal_to_Val` so as to obtain a well-defined function and use the `Define` to define the function `existtable_next`. Therefore, the `existential term` for the `TABLE` is determined by the function `existtable_next`, i.e. `existtable_next ip`.

```
∃ x1 op.  (∀ t.
      (IS_BOOL (HD ip t) ∧ IS_BOOL (HD ip t) ∧ IS_BOOL (HD (TL ip) t) ∧
      IS_BOOL (HD (TL ip) t) ∧ IS_BOOL (HD (TL (TL ip)) t)) ∧
      IS_BOOL (HD op t) ∧ IS_BOOL (HD (TL (TL op)) t) ∧
      IS_BOOL (HD (TL (TL (TL (TL op)))) t) ∧ IS_BOOL (HD (TL op) t) ∧
      IS_BOOL (HD (TL (TL (TL op))) t)) ⊃
    TABLE [x1; HD ip; HD (TL ip); HD (TL (TL ip))] (x1 o NEXT)
      [[TABLE_VAL (CONCRETE "RESET"); TABLE_VAL (BOOL T); DONT_CARE;
        DONT_CARE];
       [TABLE_VAL (CONCRETE "RESET"); TABLE_VAL (BOOL F); DONT_CARE;
        DONT_CARE];
       [TABLE_VAL (CONCRETE "COIN"); DONT_CARE; TABLE_VAL (BOOL T);
        DONT_CARE];
       [TABLE_VAL (CONCRETE "COIN"); DONT_CARE; TABLE_VAL (BOOL F);
        DONT_CARE];
       [TABLE_VAL (CONCRETE "CHANGE"); DONT_CARE; DONT_CARE;
        TABLE_VAL (BOOL T)];
        [TABLE_VAL (CONCRETE "CHANGE"); DONT_CARE; DONT_CARE;
        TABLE_VAL (BOOL F)];
       [TABLE_VAL (CONCRETE "CHOC"); DONT_CARE; DONT_CARE; DONT_CARE]]
        [(λt.  CONCRETE "COIN"); (λt.  CONCRETE "RESET");
        (λt.  CONCRETE "CHANGE"); (λt.  CONCRETE "COIN");
        (λt.  CONCRETE "CHOC"); (λt.  CONCRETE "CHANGE");
        (λt.  CONCRETE "RESET")] (λt.  CONCRETE "RESET") ∧
      TABLE [x1] (HD op) [[TABLE_VAL (CONCRETE "RESET")]] [(λt.  BOOL T)]
        (λt.  BOOL F) ∧
      TABLE [x1] (HD (TL (TL op))) [[TABLE_VAL (CONCRETE "COIN")]]
        [(λt.  BOOL T)] (λt.  BOOL F) ∧
      TABLE [x1] (HD (TL (TL (TL (TL op))))) [[TABLE_VAL (CONCRETE "CHANGE")]]
        [(λt.  BOOL T)] (λt.  BOOL F) ∧
      TABLE [x1] (HD (TL op)) [[TABLE_VAL (CONCRETE "CHANGE")]] [(λt.  BOOL T)]
        (λt.  BOOL F) ∧
      TABLE [x1] (HD (TL (TL (TL op)))) [[TABLE_VAL (CONCRETE "CHOC")]]
        [(λt.  BOOL T)] (λt.  BOOL F)
```

Figure 8.7: The Existential Theorem of the Specification of the Chocolate Machine

Secondly, we need to find the `existential term` for the output `op`. The connected five `TABLE`s are quantified by external output `op`. Each output of a `TABLE` decides one element of the output list. Because all the outputs of the `TABLE` are signals, the `existential term` for the `TABLE`s is determined by the function `existtable`. For example, the first element of the `existential term` is defined in terms of the `TABLE` whose output is `(HD op)` and is defined by the function `existtable`, which is given below:

```
(existtable [(existtable_next ip)] [[TABLE_VAL (CONCRETE "RESET")]]
          [(λt.  BOOL T)] (λt.  BOOL F))
```

Other elements in the `existential term` list can be obtained in a very similar way. They are also defined in terms of corresponding `TABLE` and function `existtable`. Therefore, the `existential term` for the output `op` can be given below:

```
[existtable [(existtable_next ip)] [[TABLE_VAL (CONCRETE "RESET")]]
          [(λt.  BOOL T)] (λt.  BOOL F);
 existtable [(existtable_next ip)][[TABLE_VAL (CONCRETE "CHANGE")]]
          [(λt.  BOOL T)] (λt.  BOOL F);
 existtable [(existtable_next ip)][[TABLE_VAL (CONCRETE "COIN")]]
          [(λt.  BOOL T)] (λt.  BOOL F);
 existtable [(existtable_next ip)][[TABLE_VAL (CONCRETE "CHOC")]]
          [(λt.  BOOL T)] (λt.  BOOL F);
 existtable [(existtable_next ip)][[TABLE_VAL (CONCRETE "CHANGE")]]
          [(λt.  BOOL T)] (λt.  BOOL F)]
```

After stripping away the leading existentially quantified variable `x1, op` using the above terms, the existential theorem for the specification of the chocolate machine (8.3) has been proved using tactic `PROVE_EXIST_TABLE_TAC`.

Finally, the conversion theorem can be obtained by discharging the formalization theorem (8.2) and the existential theorem (8.3) from the importing theorem (8.1).

This theorem states that the implementation implies the specification.

$$\vdash_{thm} \quad \forall\ \texttt{ip}\ \ \texttt{op}.\ \ \texttt{SemProgram Choc\_Imp\_Syn ip op} \supset$$
$$\texttt{SemProgram Choc\_Spe\_Syn ip op} \tag{8.4}$$

We have translated the MDG verification result into HOL to form a traditional HOL theorem. The translation process is based on the importing theorem. In other words, the linkage between the MDG system and the HOL system is the importing theorem.

8.4 Verification of the Usability Theorems

In the previous section, we imported the MDG verification result into HOL and formed the HOL theorem. How can we ensure this theorem is usable in HOL? In this section, we will use this theorem with other HOL theorems to prove the `implementation` based usability theorem to demonstrate the use of the importing theorem.

As we mentioned at the beginning of this chapter, this example was originally used by Curzon & Blandford [24], to prove the absence of post-completion errors within the framework of a traditional hardware verification. In their work, they define a formal general user model which describes the behavior of a rational user. It specifies concrete types for the machine and user state, a list of pairs of lights and the actions associated with them, history functions that represent the possessions of the user, functions that extract the part of the user state that indicates when the user has finished and has achieved their main goal and an invariant that indicates the part of the state that the user intends to be preserved after the interaction. More details can be found in [25] [24]. The general user model for a chocolate machine is defined as `CHOC_MACHINE_USER ustate op ip` which specifies the relation between the arguments discussed above.

```
⊢def CHOC_MACHINE_USER ustate op ip =
  USER
   [(CoinLight,InsertCoin); (ChocLight,PushChoc);
    (ChangeLight,PushChange)]
  (CHOC_POSSESSIONS UserHasChoc GiveChoc CountChoc UserHasChange
    GiveChange CountChange UserHasCoin InsertCoin CountCoin)
  UserFinished
  UserHasChoc
  (VALUE_INVARIANT (CHOC_POSSESSIONS UserHasChoc GiveChoc CountChoc
    UserHasChange GiveChange CountChange
    UserHasCoin InsertCoin CountCoin))
  ustate op ip
```

The usability of a chocolate machine is defined as `CHOC_MACHINE_USABLE ustate op ip` in terms of a user-centric property. It states that if at any time, t, a user approaches the machine when its coin light is on, then they will at some time, t1, have both chocolate and change.

```
⊢def CHOC_MACHINE_USABLE ustate op ip =
  ∀ t.  ∼ (UserHasChoc ustate t) ∧
        ∼ (UserHasChange ustate t) ∧
        (UserHasCoin ustate t) ∧
        (VALUE_INVARIANT (CHOC_POSSESSIONS UserHasChoc GiveChoc
             CountChoc UserHasChange GiveChange CountChange
             UserHasCoin InsertCoin CountCoin) ustate t) ∧
         ((CoinLight op t)= BOOL T) ⊃
            ∃ t1.  (UserHasChoc ustate t1) ∧
                          (UserHasChange ustate t1)
```

The `specification` based usability theorem states that if a user acts reactively and the machine behaves according to its specification, then the usability property will hold. As a matter of fact, this theorem has been proved in [25]. However, we can not make use of the usability theorem directly because the specification of the chocolate machine is different and the new type has to be defined to accommodate the different sorts. In MDG, the specifications must be in the form of a finite state machine or table description. However, the advantage of it is its speed. In HOL, the formalization is more flexible and reasonable. It need not deal with extra stuff although it might slow hardware verification.

Using our method, we have to prove a slightly different usability theorem in HOL. In the syntax of the MDG-HDL program, we use a new type `Mdg_Basic`, defined in Chapter 4, to represent the concrete type and boolean value. This is because the inputs of a `TABLE` could be either a concrete type variable or a boolean value variable. Since all the inputs and outputs of the chocolate machine are boolean values, we add additional conditions in the usability theorem to specify this fact. Hence, the usability theorem asserts the usability of an abstract specification of a chocolate machine as proved below.

```
⊢thm ∀ ustate op ip.
     (∀ t.  IS_BOOL ((HD op) t) ∧
            IS_BOOL ((HD (TL op)) t) ∧
            IS_BOOL ((HD (TL (TL op))) t) ∧
            IS_BOOL ((HD (TL (TL (TL op)))) t) ∧
            IS_BOOL ((HD (TL (TL (TL (TL op))))) t) ∧
            IS_BOOL ((HD ip) t) ∧ IS_BOOL ((HD (TL ip)) t) ∧
            IS_BOOL ((HD (TL (TL ip))) t)) ∧
        CHOC_MACHINE_USER ustate op ip ∧
              CHOC_MACHINE_SPEC ip op ⊃
                     CHOC_MACHINE_USABLE ustate op ip              (8.5)
```

Therefore, the main differences are that we need to add assumptions so as to avoid the sort of each external variable being mismatched and to ensure the specifications are in the form of a finite state machine. In practice, we can formalize the design according to this requirement at the very beginning. Although the formalization of a design is a little bit harder than the formalization of it directly in HOL, the MDG proof is quicker than HOL proof. In other words, we have to pay the price for the speed.

In the last section, we proved the correctness of the chocolate machine by using the MDG system, and formally imported it into HOL to form a HOL theorem. This theorem states that the implementation meets its specification (8.4). We also prove the `specification` based usability theorem (8.5) in HOL. The `implementation` based usability theorem can be proved in terms of the above two theorems (8.4)(8.5). This theorem (8.6) states that if the inputs and outputs are boolean value, a user acts rationally according to the user model and the machine behaves according to its `implementation`, then the usability property will hold.

$$
\begin{array}{l}
\vdash_{thm} \forall\ \texttt{ustate op ip.} \\
\quad \big(\forall\ \texttt{t.}\ \ \texttt{IS\_BOOL ((HD op) t)} \wedge \\
\qquad\quad \texttt{IS\_BOOL ((HD (TL op)) t)} \wedge \\
\qquad\quad \texttt{IS\_BOOL ((HD (TL (TL op))) t)} \wedge \\
\qquad\quad \texttt{IS\_BOOL ((HD (TL (TL (TL op)))) t)} \wedge \\
\qquad\quad \texttt{IS\_BOOL ((HD (TL (TL (TL (TL op))))) t)} \wedge \\
\qquad\quad \texttt{IS\_BOOL ((HD ip) t)} \wedge \\
\qquad\quad \texttt{IS\_BOOL ((HD (TL ip)) t)} \wedge \\
\qquad\quad \texttt{IS\_BOOL ((HD (TL (TL ip))) t)}\big) \wedge \\
\qquad \texttt{CHOC\_MACHINE\_USER ustate op ip} \wedge \\
\qquad\quad \texttt{CHOC\_MACHINE\_IMPL ip op} \supset \\
\qquad\qquad\quad \texttt{CHOC\_MACHINE\_USABLE ustate op ip}
\end{array}
\tag{8.6}
$$

From this example, we have shown that a system can be verified in two parts. One part of proof can be done in MDG, the other part of the proof can be done in HOL. The division allows MDG to be used when it would be easier than obtaining the result directly in HOL. We have provided a formal linkage between the MDG system and the HOL system, which allows the MDG verification results to be formally imported into HOL to form the HOL theorem. We do not simply assume that the results proved by MDG are directly equivalent to the result that would have been proved in HOL. The linkage is based on the importing theorems being given a greater degree of trust. We have made use of the importing theorem. In other words, the MDG verification result not only can be imported into HOL to form the HOL theorem, it also can be used as part of hierarchical hardware verification proof in HOL. We have also shown that two different applications (hardware verification and usability verification) suited to two different tools can be combined together.

However, for importing the MDG verification result into HOL, we need to prove the `existential theorem` for the specification of the design. The behaviour specifications must be in the form of a finite state machine or table description.

Summary

In this chapter, we have proved the usability theorem of a chocolate machine to demonstrate the feasibility of our methodology. We have verified the correctness of the chocolate machine in MDG, and this result has been imported into HOL to form the HOL theorem. We have proved the `specification` based usability theorem in HOL. By using the importing theorem and `specification` based usability theorem, we obtain the `implementation` based usability theorem.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

In this thesis, we have produced a methodology which can provide a formal linkage between the symbolic state enumeration system and the theorem proving system based on a verified symbolic state enumeration system. The methodology involves the following three steps.

First, we verify aspects of correctness of the symbolic state enumeration system in an interactive theorem proving system. In fact, some symbolic state enumeration based systems, such as MDG, consist of a series of translators and a set of algorithms. We need to verify the translators and algorithms to ensure the correctness of the whole system. For verifying the translators, we need to define the deep embedding semantics and translation functions. We have to make certain that the semantics of a program is preserved in its translated form. This work greatly increases the degree of trust of the symbolic state enumeration system.

Secondly, we prove importing theorems in the theorem proving system about the results from the symbolic state enumeration system. We need to formalize the correctness results produced by different hardware verification applications using the theorem proving system. The formalization is based on the semantics of the low level language (decision graph). We need to prove a theorem in each case that translates them into a form usable in the theorem proving system. In other words, we have to provide the theoretical justification for linking two systems.

Thirdly, we combine the translator correctness theorems with importing theorems. This combination allows the verification results from the state enumeration system to be formalized in terms of the semantics of a low level language (decision graph) and imported in terms of the semantics of a high level language (HDL). Therefore, we are able to import the result into the theorem proving system based on the semantics of the input language of a verified symbolic state enumeration system. This makes formalization, importation and verification easier, more direct and trustworthy.

We have also summarized a general method to prove the `existential theorem` of the design, which is needed for importing the sequential verification results into the theorem proving system. This work makes the linking process easier and remove the burden from the user of the hybrid system.

We have partly implemented this methodology in two simplified versions of the MDG system (the boolean subset and the extended subset) and the HOL system, and provide a formal linkage by using the above mentioned steps.

The standard approach of proving a translator has been used to prove the aspects of correctness of the MDG system using the HOL system. For the boolean subset, we have proved that two translators are correct (Figure 1.5). The syntax of the MDG-HDL language, the core MDG-HDL language and the MDG formula representation language have been defined in higher order logic. The semantic functions are defined by structural induction over their syntactic structure. The translation functions that translate the syntax of an MDG-HDL program to the syntax of the core MDG-HDL language and translate the syntax of the core MDG-HDL program to the syntax of the MDG formula representation language have been defined. The correctness theorem ((3.1)(3.2)) for each translator, which quantifies over its syntactic structure, has been verified. By combining these two correctness theorems we obtain a new theorem (3.3). This theorem states that the semantics of the original MDG-HDL program is equivalent to the semantics of the MDG formula representation program used in the MDG implementation.

For the extended subset, we have extended our formalization to accommodate a list of inputs of the `TABLE` component with boolean sorts and concrete sorts. We have proved that the first translator is correct (Figure 1.5). Similarly, the formal syntax and semantics of the MDG-HDL language and core MDG-HDL language of this subset has been defined. A set of functions for translating this subset language to their core MDG-HDL equivalence has then been given. The correctness theorem about the translation, which quantifies over its syntactic structure, has be proved.

In doing such a translator verification, we do more than just to prove the correctness of the system, but also build a solid foundation to formally import the MDG verification results into HOL to form the HOL theorem in terms of MDG-HDL. Our semantics of the program is represented explicitly with the external inputs and outputs, which allows the semantic function to be used in the importing theorems.

We have formally proved the general importing theorems for three different hardware verification applications using HOL. We have in each case proved a theorem that translates them into a form usable in a traditional HOL hardware verification, i.e., that the structural specification implements the behavioral specification. The first applications considered were the checking of input-output equivalence of two combinational circuits. The next application considered was sequential verification, which checks that two abstract state machines produce the same sequence of outputs for every sequence of inputs. Finally, we considered a general form of the checking of invariant properties of a circuit. These theorems are very general because they do not explicitly deal with the MDG-HDL semantics or multiway decision graph. They are given in terms of general relations on inputs and outputs. Thus they are applicable to other verification systems with a similar architecture based on reachability analysis, equivalence checking and/or invariant checking. This could include a pure BDD based system.

The two general importing theorems for each subset, combinational verification and sequential verification, have been instantiated for the semantics of the low level language. In theory, the formalization of the MDG verification result should be in terms of the MDG decision graph. However, we just proved some translators. In order to demonstrate the combination of the translator correctness theorems and the importing theorems, the formalization of the MDG results we considered here is in terms of the MDG formula representation (see Figure 6.1) for the boolean and the core MDG-HDL for the extended subset. We have combined the translator correctness theorems with the importing theorems. The combination allows the low level formalization of the MDG verification results to be imported into HOL to form the HOL theorems in terms of the semantics of MDG-HDL and the existential theorem for sequential verification to be proved in terms of the semantics of MDG-HDL. In other words, we have obtained the different theorems for two different MDG applications which explicitly deal with the MDG-HDL semantics. We thus obtain theorems that convert the low level results, which actually proved in the MDG system, to results about circuits in the high level languages in a form that can be reasoned about in HOL.

For ease of importing of MDG results into HOL for sequential verification and also for avoiding an inconsistent model, we summarize a general way to prove the existential theorem for the implementation or specification of designs based on the syntax and the semantics of MDG-HDL. We have defined the `output representation` for each component in the MDG component library. The `existential term` of a design, which strips away the leading existentially quantified variable and substitutes `term` for each free occurrence in the body, is determined in terms of those `output representations`. Since we directly deal with the syntax and semantics of the MDG-HDL program, we use a tactic `EXPAND_SEMANTICS_TAC` to expand the semantics of the program (design) and obtain a HOL goal of the form $\exists$ `a1 ... an. body`. The `existential term` can then be used to strip away the existentially quantified variable and substitute `term` for each free occurrence in the body. Two further tactics `PROVE_EXIST_TAC` and `PROVE_TABLE_EXIST_TAC` are used to solve the goal which strips away the existentially quantified variables. Although we concentrate on proving the existential theorem for the specification and implementation of a design based on the syntax and semantics of MDG-HDL, our methods can be used to solve other HOL goals which are existentially quantified. In other words, our `existential terms` and `output representations` can be used to solve existentially quantified HOL goals in other applications.

An example, the verification of correctness and usability theorems of a vending machine, has demonstrated the feasibility of our method. We have verified the correctness of the chocolate machine in MDG. The verification result has been imported into HOL to form the HOL theorem. We have proved the specification based usability theorem in HOL. By using the importing theorem and specification based usability theorem, we obtain the implementation based usability theorem.

From this example, we have shown that our method supports the hierarchical hardware verification approach as we mentioned in section 1.3.2. The MDG verification results can be fitted naturally within the HOL framework with great security using the importing theorem. We have used the importing theorem in verifying a property of a system. In other words, the MDG verification result not only can be imported into HOL to form the HOL theorem, it also can be used as part of hierarchical hardware verification proof in HOL. Furthermore, we have shown that two different applications (hardware verification and usability verification) suited to two different tools can be combined together. However, for importing the MDG verification result into HOL, we need to prove the `existential theorem` for the specification of the design.

The main difficulty we encountered is the formalization of the `TABLE`. This is because the inputs could be of different types. As a result, the formalization of a design is more complex than the formalization of it direct in HOL. This experience tells us when we design a new tool, the designers should try their best to make the tool easy to be proved at the very beginning.

9.2 Future work

We have provided a formal linkage between MDG and HOL based on a trusted MDG system. There are many opportunities for further work on verifying the correctness of the MDG system and building a verified linkage between MDG and HOL.

- `Verify the MDG algorithms`. In MDG, a set of the MDG algorithms is used to manipulate the MDGs. If the correctness theorems of the algorithms have been proved, the degree of trust of the system will increase considerably and the importing theorems which is based on the high level language (MDG-HDL) will be more reliable. Chou and Peled [17] have verified a partial-order reduction technique for model checking. Similar methods can be used to verify the MDG algorithms.
- `Verify the translators`. We have proved the translators from the MDG-HDL language to the MDG formula representation language for the boolean subset and have proved the translator from the MDG-HDL language to the core MDG-HDL language for the extended subset. Similar verifications can also be done for other translation, such as from the MDG formula representation to MDG for the boolean subset and from the core MDG-HDL to MDG for the extended subset. The more translators have been proved, the higher the degree of trust the system will have. Of course we need to use the deep embedding semantics of the corresponding language in HOL and to define the translation functions between the languages.
- `Verifying the MDG implementation`. We split the problem of verifying the translator into two problems of verifying that the implementation meets a

functional specification, and that the functional specification meets the requirement of preserving semantics. This split was advocated by Chirica and Martin [16] with respect to compiler correctness. We are concerned with the latter step here. We are not verifying the actual MDG implementation. Our formalization of the translator is a specification of it. Once combined with the translators from the core MDG-HDL to MDGs or from the MDG formula representation to MDGs, it would be specifying the output required from the implementation. It is possible to verify the MDG implementation based on the compiler specification theorems.

- `Expanding the subset language to the whole language`. The subset language we considered here did not consider three MDG predefined components (Multiplexer, Drivers and Constant) and the Transform construct used to apply functions. These components are omitted from our subset as they have non-boolean inputs or outputs. Furthermore, the subset considered does not include abstract sorts. It is possible to extend the subset to the whole language.

- `Making a linkage between two different specifications`. In MDG, the specifications must be in the form of a finite state machine or table description. This is not very abstract. The advantage of HOL is that it allows much more abstract specification. The complex MDG specification might lead to difficulty in the HOL proof. Since two different specifications formalize the same design, it may be possible to investigate the feasibility of proving the equivalent of two specifications. Or it is possible to write the tactics to simplify the MDG specification.

- `The importing theorems for model checking.` We have formally proved the general importing theorems for three different hardware verification applications using HOL. These were the original MDG tools. More recently a model checking tool was added [84]. The importing theorems for model checking can be obtained using the similar method.

- `Making use of our importing theorems with MDG-HOL`. Our importing theorems have built a solid theoretical underpinning for the linkage of HOL and MDG. It can be used in MDG-HOL or another combined system. Indeed, the MDG-HOL system shallowly embeds the semantics of MDG-HDL into HOL. It is possible to use our deep embedding semantics instead of the shallow embedding semantics so as to make use of our importing theorems.

- `Applying the methodology to a BDD based tool and theorem prover`. Our methodology works for the MDG system and the HOL system which greatly increase the degree of trust of the linkage between the two systems. Similar work can be applied to other similar automated verification tools and theorem proving systems.

Summary

The contribution of this thesis is that we have produced a methodology which can provide a formal linkage between a symbolic state enumeration system and a theorem proving system based on a verified symbolic state enumeration system. The methodology has been partly realized in two simplified versions of the MDG system and the HOL system. We have verified aspects of correctness of two simplified versions of the MDG system. We have provided a formal linkage between the MDG system and the HOL system based on importing theorems. We have combined the translator correctness theorems with the importing theorems. This combination allows the low level MDG verification results to be imported into HOL in terms of the semantics of a high level language (MDG-HDL). We have also summarized a general method which is used to prove the `existential theorem` for the specification and implementation of the design. The feasibility of this approach has been demonstrated in a case study: the verification of the correctness and usability theorems of a vending machine.

Bibliography

[1] M. D. Aagaard, R. B. Jones, R. Kaivola, and C. J. H. Seger. Formal verification of iterative algorithms in microprocessors. *DAC*, June 2000.

[2] M. D. Aagaard, R. B. Jones, and C. H. Seger. Lifted-FL: A pragmatic implementation of combined model checking and theorem proving. In *Theorem Proving in Higher Order Logics*, number 1690 in Lecture Notes in Computer Science, pages 323–340. Springer-Verlag, September 1999.

[3] M. D. Aagaard and C. J. H. Seger. The formal verification of a pipelined doubleprecision IEEE floating-point multiplier. *ICCAD, IEEE Comp. Soc.*, pages 7–10, November 1995.

[4] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, c-27(6):509–516, June 1978.

[5] P. Argon and K. McMillan. Deriving a special-purpose prover for compositional model checking in Coq. In *TPHOLs 2000 Supplemental Proceedings*, pages 1–5. Oregon Graduate Institute, 2000.

[6] G. Birtwistle, S. Chin, and B. Graham. *new_theory 'HOL';; An Introduction to Hardware Verification in Higher Order Logic.* Unpublished, 1994. http://www.comp.leeds.ac.uk/graham/research/hv/hvbooks.html.

[7] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. Van-Tassel. Experience with embedding hardware description language in HOL. In T. F.

Melham and R. T. Boute, editors, *Theorem Provers in Circuit Design*, pages 129–156. North-Holland, 1992.

[8] R. S. Boyer and G. Dowek. Towards checking proof checkers. In *Workshop on Types for Proofs and Programs (Type'93)*, 1993.

[9] R. S. Boyer and J. Moore. *A Computational Logic Handbook*. Academic Press, London, 1997.

[10] B. C. Brock and W. A. Hunt. The formalization of a simple hardware description language. In Luc Claesen, editor, *Applied Formal Methods for Correct VLSI Design*, pages 778–792, Amsterdam, November 1989. IMEC-IFIP International Workshop, Elsevier Science Publishers.

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

[12] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computer Surveys*, 24(3), September 1992.

[13] R. S. Burstall and P. J. Landin. Programs and their proofs: an algebraic approach. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence*, number 4, pages 17–43. Edinburgh University Press, 1969.

[14] A. Camilleri, M. Gordon, and T. Melham. Hardware verification using Higher-Order Logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs: Proceedings of the IFIP WG 10.2 Working Conference*, pages 43–67, Grenoble, September 1986.

[15] L. M. Chirica. *Contributions to Compiler Correctness*. Number Report UCLA-ENG-7697. Computer Science Department, University of California, Los Angeles, October 1976. Ph.D. thesis.

[16] L. M. Chirica and D. F. Martin. Toward compiler implementation correctness proofs. *ACM Transactions on Programming Languages and Systems*, 8(2):185–214, April 1986.

[17] C. T. Chou and D. Peled. Formal verification of a partial-order reduction technique for model checking. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, number 1055 in Lecture Notes in Computer Science, pages 241–257, 1996.

[18] A. Cohn and R. Milner. On using Edinburgh LCF to prove the correctness of a parsing algorithm. Technical Report 20, University of Edinburgh Computer Science, 1982.

[19] P. A. Collier. Simple compiler correctness - a tutorial on the algebraic approach. *The Australian Computer Journal*, 18(3), August 1986.

[20] F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny. Multiway decision graphs for automated hardware verification. *Formal Methods in System Design*, 10(1):7–46, 1997.

[21] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. http://www.dcs.gla.ac.uk/prosper/papers.html, 1999.

[22] P. Curzon. A verified Vista implementation. Technical Report 311, University of Cambridge, Computer Laboratory, September 1993.

[23] P. Curzon. The formal verification of the Fairisle ATM switching element. Technical Report 329, University of Cambridge, Computer Laboratory, March 1994.

[24] P. Curzon and A. Blandford. Using a verification system to reason about post-completion errors. In *Participants Proceedings of DSV-IS 2000: 7th International Workshop on Design, Specification and Verification of Interactive Systems, at the 22nd International Conference on Software Engineering.*

[25] P. Curzon and A. Blandford. Reasoning about order errors in interaction. In *TPHOLs 2000 Supplemental Proceedings*, Technical Reprot CSE-00-009, pages 33–48. Oregon Graduate Institute, August 2000.

[26] P. Curzon, S. Tahar, and O. Aït-Mohamed. Verification of the MDG components library in HOL. In Jim Grundy and Malcolm Newey, editors, *Theorem Proving in Higher-Order Logics: Emerging Trends*, pages 31–46. Department of Computer Science, The Australian National University, 1998.

[27] L. A. Dennis, G. Collins, M. Norrish, R. Boulton, K. Slind, G. Robinson, M. Gordon, and T. Melham. The PROSPER toolkit. http://www.dcs.gla.ac.uk/prosper/papers.html, 1999.

[28] Computer General Electronic Design. *The ELLE Language Reference Manual, Issue 4.0.* Greenways Business Park, Bellinger Close, Chippenham, Wiltshire, SN15 1BN, England, 1989.

[29] D. I. Good, R. L. Akers, and L. M. Smith. Report on Gypsy 2.05. Technical Report CLI-1, Computational Logic, Inc., 1986.

[30] K. Goossens. *Embedding Hardware Description Languages in Proof Systems.* Laboratory for Foundations of Computer Science, Department of Computing Science , University of Edinburgh, December 1992. Ph.D. thesis.

[31] M. J. Gordon. Synthesizable verilog syntax and semantics. Technical report, University of Cambridge, Computer Laboratory, January 1997. www.cl.cam.ac.uk/users/mjcg/V/V.html.

[32] M. J. Gordon. Notes on the representation of state machines in higher order logic. Technical report, University of Cambridge, Computer Laboratory, January 1999.

[33] M. J. Gordon, T. Kropf, and D. Hoffmann. PROSPER ESPRIT LTR project 26241, semantics of the intermediate language IL. Technical report, University of Cambridge, Computer Laboratory, February 1999.

[34] M. J. Gordon, R. Milner, and C. P. Wadsworth. Edinburgh LCF: A mechanised logic of computation. Number 78 in Lecture Notes in Computer Science, 1979.

[35] M. J. C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. J. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design: the 1985 Edinburgh Workshop on VLSI*, pages 153–177. North-Holland, 1986.

[36] M. J. C. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic, 1988.

[37] M. J. C. Gordon. Mechanizing programming logics in higher order logic. In P. A. Subrahmanyam and G. Birtwistle, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, number 7, pages 387–489, New York, 1989. Springer-Verlag.

[38] M. J. C. Gordon. Combining deductive theorem proving with symbolic state enumeration. Presented at 21 Years of Hardware Verification, Royal Society Workshop to mark 21 years of BCS FACS, http://www.cl.cam.ac.uk/users/mjcg/BDD, December 1998.

[39] M. J. C. Gordon. Reachability programming in HOL98 using BDDs. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics*, number 1869 in Lecture Notes in Computing Science, pages 179–196. Springer-Verlag, Aug. 2000.

[40] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-order Logic*. Cambridge University Press, 1993.

[41] E. L. Gunter and D. Obradovic. Towards the integration of model checking and theorem proving: Embedding a subset of Promela into HOL. In *TPHOLs 2000 Supplemental Proceedings*, Technical Reprot CSE-00-009, pages 75–85. Oregon Graduate Institute, August 2000.

[42] J. Harrison and L. Théry. A skeptic's approach to combining HOL and Maple. *Journal of Automated Reasoning*, 21:279–294, 1998.

[43] S. Hazelhurst and C. J. H. Seger. A simple theorem prover based on symbolic trajectory evaluation and BDDs. *IEEE Trans. on CAD*, April 1995.

[44] S. Hazelhurst and C. J. H. Seger. *Symbolic trajectory evaluation.* Springer Verlag. New York, 1997.

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

[46] P. V. Homeier and D. F. Martin. A verified verification condition generator. *The Computer Journal*, 38(2):131–141, July 1995.

[47] A. Hu. Formal hardware verification with BDDs: An introduction. In *IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing (PACRIM)*, pages 667–682, 1997.

[48] G. Huet, G. Kahn, and C. Paulin-Mohring. The Coq proof assistant - a tutorial, version 6.1. Technical Report 204, INRIA, August 1997.

[49] J. Hurd. Integrating GANDALF and HOL. Technical Report 461, University of Cambridge, Computer Laboratory, April 1999.

[50] J. Joyce. A verified compiler for a verified microprocessor. Technical Report 167, University of Cambridge, Computer Laboratory, March 1989.

[51] J. Joyce and C. Seger. Linking BDD-based symbolic evaluation to interactive theorem-proving. In *the 30th Design Automation Conference*, 1993.

[52] R. Kaivola and M. D. Aagaard. Divider circuit verification with model checking and theorem proving. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics*, number 1869 in Lecture Notes in Computer Science, 13 International Conference, TPHOLs 2000, Portland, OR, USA, August 2000. Springer-Verlag.

[53] S. Kort, S. Tahar, and P. Curzon. Hierarchical hardware verification using a hybrid tool. Technical report, Dept. of Electrical and Computer Engineering,

Concordia University, 1455 De Maisonncuve West, Montreal, Quebee - H3G LM8, Canada, 2000.

[54] S. Kort, S. Tahar, and P. Curzon. Hierarchical verification using an MDG-HOL hybrid tool. In T. Margaria and T. Melham, editors, *11th IFIP WG 10.5 Advanced Research Working Conference (CHARME'2001)*, number 2144 in Lecture Notes in Computer Science, pages 244–258, Livingston, Scotland, UK, September 2001. Springer-Verlag.

[55] T. Kropf and R. Reetz. Simplifying deep embedding: A formalised code generator. In J. Camilleri and T. Melham, editors, *Higher Order Logic Theorem Proving and its Applications*, number 859 in Lecture Notes in Computer Science. Springer-Verlag, September 1995.

[56] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In J. Schwartz, editor, *A Symposium on Applied Mathematics*, pages 33–41, 1967.

[57] T. F. Melham. Automating recursive type definitions in Higher Order Logic. In *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386. Springer Verlag, 1989.

[58] T. F. Melham. *Higher Order Logic and Hardware Verification.* Cambridge Tracts in Theoretical Computer Science 31. Cambridge University Press, 1993.

[59] R. Milner and R. Weyhrauch. Proving compiler correctness in a mechanized logic. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence*, number 7, pages 51–70, Edinburgh, Scotland, 1972. Edinburgh University Press.

[60] J. Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, (5):461–492, 1989.

[61] J. S. Moore. A mechanically verified language implementation. Technical Report CLI-22, Computational Logic, Inc., 1988.

[62] F. L. Morris. *Correctness of Translations of Programming Languages.* Report STAN-CS-72-303. Computer Science Department, Stanford University, August 1972. Ph.D. thesis.

[63] F. L. Morris. Advice on structure compilers and proving theorem correct. In *The ACM Symposium on Principles of Programming Languages*, pages 144–152, Boston, October 1973.

[64] Institute of Electrical and Electronics Engineers. *IEEE standard VHDL language Reference Manual.* IEEE press. New York, 1988.

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

[66] V. K. Pisini and S. Tahar. Integration of HOL and MDG for hardware verification. Technical report, Dept. of Electrical and Computer Engineering, Concordia University, 1455 De Maisonncuve West, Montreal, Quebee - H3G LM8, Canada, March 1999.

[67] V. K. Pisini, S. Tahar, P. Curzon, and O. Ait-Mohamed. A hybrid approach to formal verification using HOL and MDG. Technical report, Dept. of Electrical and Computer Engineering, Concordia University, 1455 De Maisonncuve West, Montreal, Quebee - H3G LM8, Canada, November 1999.

[68] G. Pottinger. Completeness for the HOL logic: Preliminary report. In *Posted to info-hol mail list on 28th Jan 1992.*, 1992. Available in the info-hol archive by anonymous FTP from ftp.cl.cam.ac.uk in directory hvg/info-hol-archive.

[69] S. Rajan, N. Shankar, and M. K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification*, number 939 in Lecture Notes in Computer Science, pages 84–97. Springer-Verlag, 1995.

[70] K. Schneider and T. Kropf. Verifying hardware correctness by combining theorem proving and model checking. Technical Report SBF 358-C2-5/95, University of Karlsruhe, Department of Computer Science, 1995.

[71] K. Schneider and T. Kropf. Unified approach for combining different formalisms for hardware verification. Technical Report SBF 358-C2-6/96, University of Karlsruhe, Department of Computer Science, January 1996.

[72] C.-J. H. Seger and R. E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2):147–190, March 1995.

[73] S. Tahar, X. Song, E. Cerny, Z. Zhou, M. Langevin, and O. Aït-Mohamed. Modeling and automatic formal verification of the Fairisle ATM switch fabric using MDGs. To appear in *IEEE Transactions on CAD of Integrated Circuits and Systems.*

[74] J. von Wright. Program refinement by theorem prover. In *Proc. 6th Refinement Workshop*, London, January 1994. Springer–Verlag.

[75] J. von Wright. Representing higher-order logic proofs in HOL. *The Computer Journal*, 38(2):171–179, July 1995.

[76] J. von Wright. The formal verification of a proof checker. SRI internal report, November 1998.

[77] W. Wong. Validation of HOL proofs by proof checking. *Formal Methods in System Design*, 14(2):193–212, March 1999.

[78] H. Xiong and P. Curzon. The verification of a translator for MDG's components in HOL. In *MUCORT98, Third Middlesex University Conference on Research in Technology*, pages 55–59, April 1998.

[79] H. Xiong, P. Curzon, and A. Blandford. Combining verification systems in a trusted way to reap the benefits of both. In *Automated Reasoning-Bridging the Gap between Theory and Practice The 6th Workshop*, pages 71–73, April 1999.

[80] H. Xiong, P. Curzon, and S. Tahar. Importing MDG verification results into HOL. In *Theorem Proving in Higher Order Logics*, number 1690 in Lecture Notes in Computer Science, pages 293–310. Springer-Verlag, September 1999.

[81] H. Xiong, P. Curzon, S. Tahar, and A. Blandford. Verification of a translator for MDG's library in HOL. In *15th British Colloquium for Theoretical Computer Science*, April 1999.

[82] H. Xiong, P. Curzon, S. Tahar, and A. Blandford. Embedding and verification of an MDG-HDL translator in HOL. In *TPHOLs 2000 Supplemental Proceedings*, Technical Reprot CSE-00-009, pages 237–248. Oregon Graduate Institute, August 2000.

[83] H. Xiong, P. Curzon, S. Tahar, and A. Blandford. Proving existential theorems when importing results from MDG to HOL. In Richard J. Boulton and Paul B. Jackson, editors, *TPHOLs 2001 Supplemental Proceedings*, Informatic Research Report EDI-INF-RR-0046, pages 384–399. Division of Informatics, University of Edinburgh, Edinburgh, UK, September 2001.

[84] Y. Xu. *Model Checking for a Forst-order Temporal Logic Using Multiway Decision Graphs*. 1455 De Maisonncuve West, Montreal, Quebee - H3G LM8, Canada, 1999. Ph.D. thesis.

[85] W. D. Young. A mechanically verified code generator. *Journal of Automated Reasoning*, (5):493–519, 1989.

[86] Z. Zhou and N. Boulerice. *MDG Tools (V1.0) User Manual*. University of Montreal, Dept. D'IRO, 1996.

[87] Z. Zhu, J. Joyce, and C. Seger. Verification of the Tamarack-3 microprocessor in a hybrid verification environment. In *Higher-Order Logic theorem proving and Its Applications, The 6th International Workshop*, number 780 in Lecture Notes in Computer Science, pages 252–266. B. C., Canada, August 1993.

Appendix A

The Abstract Syntax of a Boolean Subset

The full abstract syntax of the boolean subset of the MDG-HDL language is given below:

```
out_type ::= NOWV of string |
             NEXTV of string

default_type ::= DENORMAL of num->bool |
                 DEOUT of out_type |
                 DECONST of string

Table_Val ::= TABLE_VAL of α | DON'T_CARE

mdg_hdl ::= NOT of string=>string |
            AND of string=>string=>string |
            OR of string=>string=>string |
            NAND of string=>string=>string |
            XOR of string=>string=>string |
```

```
        NOR of string=>string=>string |
        AND3 of string=>string=>string=>string |
        OR3 of string=>string=>string=>string |
        NAND3 of string=>string=>string=>string |
        NOR3 of string=>string=>string=>string |
        AND4 of string=>string=>string=>string=>string |
        OR4 of string=>string=>string=>string=>string |
        NAND4 of string=>string=>string=>string=>string |
        NOR4 of string=>string=>string=>string=>string |
        AND5 of string=>string=>string=>string=>string=>string |
        OR5 of string=>string=>string=>string=>string=>string |
        NAND5 of string=>string=>string=>string=>string=>string |
        NOR5 of string=>string=>string=>string=>string=>string |
        AND6 of string=>string=>string=>string=>string=>string=>string |
        OR6 of string=>string=>string=>string=>string=>string=>string |
        NAND6 of string=>string=>string=>string=>string=>string=>string |
        NOR6 of string=>string=>string=>string=>string=>string=>string |
        JKFF of string=>string=>string |
        RSFF of string=>string=>string |
        JKFFE of string=>string=>string=>string |
        AO of string=>string=>string=>string=>string |
        REGCON of string=>string=>string |
        REG of string=>string |
        FORK of string=>string |
        INIT of (string#bool) |
        SNXT of string=>string |
        TABLESYN of (string list)=>out_type=>((bool Table_Val list) list)
                     =>((num->bool) list)=>default_type |
        JOIN of mdg_hdl=>mdg_hdl

Exoutput ::= EXOUT of string list
Exinput ::= EXIN of string list
Invariable ::= INV of string list
```

```
program ::= PROG of PROG of Exoutput=>Exinput=>Invariable=>Mdg_Hdl
```

Appendix B

The Abstract Syntax of an Extended Subset

The full abstract syntax of the extended subset of the MDG-HDL language is given below:

```
Out_Type ::= NOWV of string |
           NEXTV of string

Default_Type ::= DENORMAL of num->bool |
                DEOUT of out_type |
                DECONST of string

Table_Val ::= TABLE_VAL of α | DON'T_CARE

Mdg_Basic ::= UNBOUND | BOOL of bool | CONCRETE of string

Mdg_Hdl ::= NOT of string=>string |
          AND of string=>string=>string |
          OR of string=>string=>string |
```

```
NAND of string=>string=>string |
XOR of string=>strin=>string |
NOR of string=>string=>string |
AND3 of string=>string=>string=>string |
OR3 of string=>string=>string=>string |
NAND3 of string=>string=>string=>string |
NOR3 of string=>string=>string=>string |
AND4 of string=>string=>string=>string=>string |
OR4 of string=>string=>string=>string=>string |
NAND4 of string=>string=>string=>string=>string |
NOR4 of string=>string=>string=>string=>string |
AND5 of string=>string=>string=>string=>string=>string |
OR5 of string=>string=>string=>string=>string=>string |
NAND5 of string=>string=>string=>string=>string=>string |
NOR5 of string=>string=>string=>string=>string=>string |
AND6 of string=>string=>string=>string=>string=>string=>string |
OR6 of string=>string=>string=>string=>string=>string=>string |
NAND6 of string=>string=>string=>string=>string=>string=>string |
NOR6 of string=>string=>string=>string=>string=>string=>string |
JKFF of string=>string=>string |
RSFF of string=>string=>string |
JKFFE of string=>string=>string=>string |
AO of string=>string=>string=>string=>string |
REGCON of string=>string=>string |
REG of string=>string |
FORK of string=>string |
INIT of (string#Mdg_Basic) |
SNXT of string=>string |
TABLESYN of (string list)=>Out_Type=>((Mdg_Basic Table_Val list) list)
              =>((num->bool) list)=>Default_Type |
SEQ of Mdg_Hdl=>Mdg_Hdl |
INTERNAL of string => Mdg_Hdl
```

```
Exoutput ::= EXOUT of string list

Exinput ::= EXIN of string list

Invariable ::= INV of string list

Mdg_Program ::= PROG of Exoutput =>Exinput => Invariable => Mdg_Hdl
```

Appendix C

The MDG-HDL programs of the verification of the Chocolate Machine

When we verify the correctness of the chocolate machine in MDG, we need to provide four MDG-HDL files. Those files are given below:

(1). The Circuit Specification File.

```
% Multifile declaration required by Prolog system.%
:- multifile signal/2.
:- multifile component/2.
:- multifile st_nxst/2.
:- multifile next_state_partition/1.
:- multifile output_partition/1.
:- multifile outputs/1.
:- multifile init_val/2.
:- multifile init_var/2.
:- multifile par_strategy/2.
```

```
%--- Common signals ---%
signal(insertCoin,bool).
signal(pushChoc,bool).
signal(chocSt, chocStates).
signal(giveChange,bool).
signal(pushChange,bool).
signal(chocLight,bool).
signal(coinLight,bool).
signal(giveChoc, bool).
signal(changeLight,bool).
%--- Components of X ---%
component(choc_machine,
        table([[chocSt,insertCoin,pushChange,pushChoc, n_chocSt],
               [reset,1,*,*,coin],[reset,0,*,*,reset],
               [coin,*,1,*,change],[coin,*,0,*,coin],
               [change,*,*,1,choc],[change,*,*,0,change],
               [choc,*,*,*,reset]])).
component(coin_light, table([[chocSt,coinLight],[reset, 1] | 0])).
component(change_light, table([[chocSt,changeLight], [coin, 1] | 0])).
component(give_change, table([[chocSt,giveChange], [change, 1] | 0])).
component(choc_light, table([[chocSt,chocLight], [change, 1] | 0])).
component(give_choc, table([[chocSt,giveChoc], [choc, 1] | 0])).
%--- Initial state ---%
init_val(chocSt,reset).
outputs([coinLight,chocLight, changeLight,giveChoc,giveChange]).
%--- Partitions ---%
output_partition([[[coinLight]],[[chocLight]], [[changeLight]],
                  [[giveChoc]],[[giveChange]]]).
next_state_partition([[[n_chocSt]]]).
%--- State variables to next state variables mapping ---%
st_nxst(chocSt, n_chocSt).
%--- Partition strategy ---%
par_strategy(auto,auto).
```

(2). The Circuit Implementation File

```
% Multifile declaration required by Prolog system.%
:- multifile signal/2.
:- multifile component/2.
:- multifile st_nxst/2.
:- multifile next_state_partition/1.
:- multifile output_partition/1.
:- multifile outputs/1.
:- multifile init_val/2.
:- multifile init_var/2.
:- multifile par_strategy/2.
%--- Common signals ---%
signal(insertCoin,bool).
signal(pushChange,bool).
signal(pushChoc,bool).
signal(l1,bool).
signal(choc_a,bool).
signal(xin,bool).
signal(coin_a, bool).
signal(reset_a,bool).
signal(l2,bool).
signal(l4,bool).
signal(l3,bool).
signal(l5,bool).
signal(yin,bool).
signal(x,bool).
signal(givenChoc_a,bool).
signal(y,bool).
signal(xbar,bool).
signal(ybar,bool).
signal(change_a,bool).
signal(givenChange_a,bool).
```

```
signal(chocLight_a,bool).
signal(changeLight_a,bool).
signal(coinLight_a,bool).
%--- Components of X ---%
component(x_and,and(input(coin_a,pushChange),output(l1))).
component(x_or,or(input(change_a,l1),output(xin))).
%---Components of Y---%
component(y_and_rein, and(input(reset_a,insertCoin), output(l2))).
component(y_or_col2, or(input(coin_a,l2),output(l4))).
component(y_inv, not(input(pushChoc),output(l3))).
component(y_and_chl3, and(input(change_a,l3),output(l5))).
component(y_or_l4l5, or(input(l4,l5), output(yin))).
%---Component of Register--%
component(reg_x,reg(input(xin),output(x))).
component(reg_y,reg(input(yin),output(y))).
%---Component of Output from the register--%
component(outreg_inv_x,not(input(x),output(xbar))).
component(outreg_inv_y,not(input(y),output(ybar))).
component(outreg_and_xy, and(input(x,y),output(change_a))).
component(outreg_and_xybar, and(input(x,ybar),output(choc_a))).
component(outreg_and_xbary, and(input(xbar,y),output(coin_a))).
component(outreg_and_xbarybar, and(input(xbar,ybar),output(reset_a))).
%---Wire output---%
component(wire_choc_givenchoc,fork(input(choc_a),output(givenChoc_a))).
component(wire_choc_changlight,fork(input(change_a),output(chocLight_a))).
component(wire_change_givechange,fork(input(change_a),output(givenChange_a))).
component(wire_coin_choclight,fork(input(coin_a),output(changeLight_a))).
component(wire_reset_coinlight,fork(input(reset_a),output(coinLight_a))).
%--- Initial state ---%
init_val(x, 0).
init_val(y, 0).
outputs([coinLight_a,chocLight_a, changeLight_a,givenChoc_a,givenChange_a]).
%--- Partitions ---%
```

```
output_partition([[[coinLight_a]],[[chocLight_a]], [[changeLight_a]],
                   [[givenChoc_a]],[[givenChange_a]]]).
next_state_partition([[[xin]],[[yin]]]).
%--- State variables to next state variables mapping ---%
st_nxst(x, xin).
st_nxst(y, yin).
%--- Partition strategy ---%
par_strategy(auto,auto).
```

(3).The Algebraic Specification File

```
% Multifile definition for Prolog predicates.% :- multifile abs_sort/1.
:- multifile conc_sort/2.
:- multifile function/3.
:- multifile gen_const/2.
:- multifile rr/3.
:- multifile ucrr/2.
% Algebraic specification% conc_sort(chocStates, [reset, coin, choc, change]).
```

(4). The Symbol Order File

```
order_main([
insertCoin,
pushChoc,
pushChange,
%---internal---%
chocSt,
n_chocSt,
coin_a,
l1,
choc_a,
reset_a,
```

```
l2,
l4,
l3,
l5,
xin,
x,
yin,
y,
xbar,
ybar,
change_a,
%---outputs---%
giveChange,
givenChange_a,
chocLight,
chocLight_a,
coinLight,
coinLight_a,
giveChoc,
givenChoc_a,
changeLight,
changeLight_a
]).
```

(5). The Invariant Specification File

```
signal(insertCoin,bool).
signal(pushChoc,bool).
signal(pushChange,bool).
signal(coinLight,bool).
signal(coinLight_a,bool).
signal(u_CoinLight,bool).
signal(chocLight,bool).
```

```
signal(chocLight_a,bool).
signal(u_ChocLight,bool).
signal(changeLight,bool).
signal(changeLight_a,bool).
signal(u_ChangeLight,bool).
signal(giveChoc,bool).
signal(givenChoc_a,bool).
signal(u_GivenChoc,bool).
signal(giveChange,bool).
signal(givenChange_a,bool).
signal(u_GivenChange,bool).
%--- Components ---%
component(coinLight_fork1,fork(input(u_CoinLight),output(coinLight))).
component(coinLight_fork2,fork(input(u_CoinLight),output(coinLight_a))).
component(chocLight_fork1,fork(input(u_ChocLight),output(chocLight))).
component(chocLight_fork2,fork(input(u_ChocLight),output(chocLight_a))).
component(changeLight_fork1,fork(input(u_ChangeLight),output(changeLight))).
component(changeLight_fork2,fork(input(u_ChangeLight),output(changeLight_a))).
component(givenChoc_fork1,fork(input(u_GivenChoc),output(giveChoc))).
component(givenChoc_fork2,fork(input(u_GivenChoc),output(givenChoc_a))).
component(givenChange_fork1,fork(input(u_GivenChange),output(giveChange))).
component(givenChange_fork2,fork(input(u_GivenChange),output(givenChange_a))).
%--- Outputs ---%
outputs([coinLight,coinLight_a, chocLight, chocLight_a, changeLight,
    changeLight_a, giveChoc,givenChoc_a, giveChange, givenChange_a]).
%--- Order of condition signals ---%
order_cond([
insertCoin,
pushChoc,
pushChange,
u_CoinLight,
coinLight,
coinLight1,
```

```
u_ChocLight,
chocLight,
chocLight_a,
u_ChangeLight,
changeLight,
changeLight_a,
u_GivenChoc,
giveChoc,
givenChoc_a,
u_GivenChange,
giveChange,
givenChange_a]).
```