Embedding and Verification of an MDG-HDL Translator in HOL

Haiyan Xiong[1], Paul Curzon[1], Sofiène Tahar[2], and Ann Blandford[1]

[1] School of Computing Science, Middlesex University, London, UK
{h.xiong, p.curzon, a.blandford}@mdx.ac.uk
[2] ECE Department, Concordia University, Montreal, Canada.
tahar@ece.concordia.ca

Abstract. We investigate the verification of a translation phase of the Multiway Decision Graphs (MDG) verification system using the Higher Order Logic (HOL) theorem prover. In this paper, we deeply embed the semantics of a subset of the MDG-HDL language and its Table subset into HOL. We define a set of functions which translate this subset MDG-HDL language to its Table subset. A correctness theorem for this translator, which quantifies over its syntactic structure, has been proved. This theorem states that the semantics of the MDG-HDL program is equivalent to the semantics of its Table subset.

1 Introduction

The application of BDD (Binary Decision Diagram) [3] based tools in digital circuit synthesis and verification has been a breakthrough for the use of formal verification by industry. However, many questions remain about whether they work effectively or not. Ideally verification systems should themselves be formally verified using a verification system with a different architecture. Based on this consideration, we investigate the verification of aspects of the Multiway Decision Graphs (MDG) verification system [6] using the Higher Order Logic (HOL) theorem prover [9].

A variety of technologies have been used to ensure the correctness of verification systems. In a sense, which method is appropriate depends on the architecture of the verification system. The Edinburgh LCF (Logic of Computable Functions) [8] family of theorem provers (including HOL) uses an abstract data type (Thm) to represent theorems. The type checker ensures the theorems can be constructed only 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. It effectively increases the reliability of the system. In this way if we guarantee the primitive inference rules correct then invalid theorems can be avoided. Moreover, the LCF approach 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.

The architecture of a symbolic state enumeration based verification system is different. In this kind of system, 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 diagrams. A series of algorithms in the system is used to efficiently and automatically deal with the decision diagrams and obtain the correctness results. The following two aspects of the system need to be verified:

1. the correctness of translation from the higher level languages into decision diagrams, and
2. the correctness of algorithms that are used to manipulate the decision diagrams.

In this paper, we prove the correctness of the translation phase of the MDG system. We need to verify that the semantics of a program is preserved in the semantics of its translated form. In this sense, it is a similar problem to that of compiler verification [4]. The contribution of this paper is to demonstrate how compiler correctness work can be applied to a hardware verification system. In doing such a verification, we do more than just prove the correctness of the system, but also build a solid foundation to combine the HOL and MDG systems in a trusted way. Because we use a deep embedding semantics, the compiler correctness theorem can be combined with theorems converting MDG results into a form that can be easily reasoned about in HOL [16]. We thus obtain theorems that convert low level results actually proved in the core of the hardware verification systems (e.g., about decision graphs) to results about circuits in high level languages in a form that can be reasoned about in a theorem prover. We are thus able to import the MDG results into HOL based on a trusted MDG system.

The structure of this paper is as follows. In Section 2, we review related work. In Section 3, we overview the MDG verification system. In Section 4, we give the formal syntax and semantics of the subset of the MDG-HDL language we use. Here a set of functions for translating this subset language to their Table equivalent is given. Furthermore, the correctness theorem we have proved about the translation, which quantifies over its syntactic structure, is described. Finally, our conclusions and ideas for further work are presented in Section 5.

2 Related Work

There have been several previous projects concerned with the validation of results from verification systems.

Wong [15] developed a proof checker to examine the correctness of 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. Von Wright [13] formalised the specification of a proof checker in HOL. He also demonstrated how the HOL system could be used to formally verify the specification of a proof checker for higher-order logic proofs [14]. Another method of using refinement to verify the proof checker had been suggested by von Wright [12]. The proof checker also provided an independent means of ensuring the validity and consistency of proofs. Some other theorem provers such as Nqthm, Nuprl and Coq already store proof trees upon which a proof checker could work in the system. Boyer and Dowek [2] specified and implemented a proof checker in Nqthm logic.

Homeier and Martin [10] used the HOL system to verify a verification system called a verification condition generator (VCG) for a simple programming language. The proof of correctness of the VCG can be considered as an example of a compiler correctness problem, since the VCG translated the annotated programs to the lists of verification conditions. The semantics of the annotated programs and verification conditions were formalised in HOL. The correctness theorems showed that the truth of the verification conditions implied the truth of the annotated programs.

Chou and Peled [5] used the HOL system to verify a non-trivial algorithm–implementing a Partial-Order reduction technique, used in the protocol verification tool SPIN, which cuts 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 paper, we verify the translation phase of the MDG system by using HOL. We need to verify that the compiler preserves the semantics of a program through the translation between languages as suggested for Homeier and Martin's work [10]. They used compiler verification methods to verify a software verification system. We use a similar method to verify a hardware verification system–the MDG system using HOL. In our study, we deeply embed a subset of the MDG-HDL language and its Table subset in HOL and verify the correctness of the translation between these two languages. Curzon et al. [7] did some basic work which verified the MDG components library in HOL. In their work, the semantics of the TABLE was first formalised in HOL. The TABLE construct is one of the basic hardware components used to define both behavioral specifications and structural specifications. Other components such as logic gates can be defined in terms of it. They verified the Table implementations of each of the hardware components that were implemented in terms of tables in the MDG system. They used a shallow embedding semantics [1]–only the semantics is represented in the HOL logic not the syntax. Whilst this can be used to prove that each individual component implementation meets its specification, it cannot be used to give a general correctness theorem about the whole MDG-HDL language. We verify that the translation process is correct based on a deep embedding semantics [1] (i.e., we represent the abstract syntax of HDL programs by terms then define within the logic semantic functions that assign meanings to the programs). This allows us to prove a theorem that quantifies over the syntactic structure of the MDG-HDL language. That is we can prove "for all MDG-HDL programs, the semantics of a program is preserved in the semantics of its translated form".

3 The MDG System

MDG-HDL [17] is a Prolog-style hardware description language, which allows the use of abstract variables for representing data signals. 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, 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 description of high-level constructs as ITE (If-Then-Else) formulas and CASE formulas.

Most of the components have their own tabular code and are compiled into their tabular code first. Tabular code can then be compiled into an internal MDG decision graph. Some components such as registers are implemented directly in terms of MDGs. However, in theory these components also could be implemented as tables. In this paper, we defined corresponding Tables for these components (register, fork, etc.). Since we are considering only a boolean subset of the language here, the Table representation of these components can be defined in terms of the corresponding input values (true or false). These definitions can be implemented in the MDG package. Non-boolean sorts could be handled by introducing an additional variable into the table. We assume the MDG-HDL program is firstly translated into a Table program and then the Table program is translated into MDG. In this situation, the MDG system could be specified as indicated in Figure 1:

$$\text{MDH-HDL} \longrightarrow \text{TABLE} \longrightarrow \text{MDGs}$$

Fig. 1. Overview of the MDG Translation Phases

Adopting this approach makes the translation phase more amenable to verification. We are not verifying the actual MDG implementation. Rather our formalisation of the translator is a specification of it. Once combined with a translator from Tables 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 [4] with respect to compiler correctness.

4 MDG Translator Verification

The intention of our research is to explore a way of combining the MDG system and the HOL system in a trusted way as shown in Figure 2. This work can be divided into two steps. We first must verify the correctness of the MDG system using the HOL system (1) based on the semantics of the MDG input language. This part of the work consists of two phases–(1a) verification of the translator and (1b) verification of the algorithms. (2) We then must verify the HOL theorem generator which formalises the MDG verification results of different MDG applications and then converts them into the traditional HOL hardware verification theorems. All of these are based on the deep embedding of the semantics. By combining the separate correctness theorems from these two steps, we obtain a result that justifies the use of "theorems" imported from MDG [17].

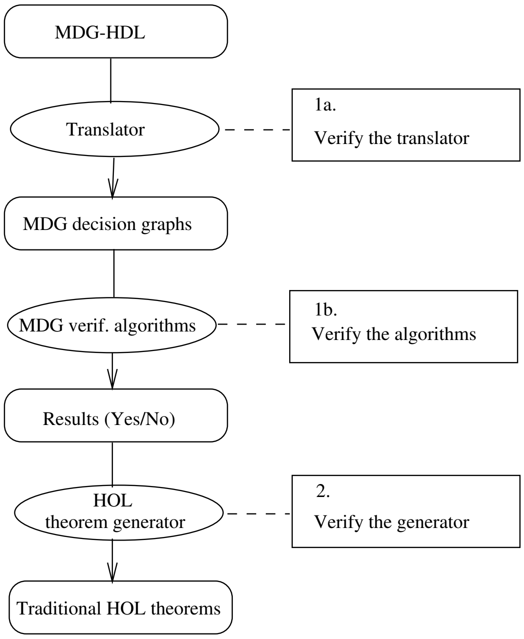

Fig. 2. Combining MDG and HOL in a Trusted Way

During this study, we considered a subset of the MDG-HDL language that did not contain two MDG predefined components (Multiplexer and Drivers) and nor do we consider the Transform construct used to apply functions. These components were omitted from our initial subset as they have non-boolean inputs or outputs. We also assume that the inputs and outputs of each component had Boolean sorts. We keep the subset simple here since we want to explore the feasibility of this method. However, we could extend our formalisation to accommodate different types as explained in [7]. As a result, the syntax of this language will be more complex. To distinguish between our subset of the MDG-HDL language and the Table subset, in the rest of this paper we will refer to the Table subset as the Table language.

In this paper, we concentrate on the verification of the translation phase of the MDG system (step (1a) from Figure 2) based on the semantics of the MDG input language using the HOL theorem prover. Step 2 is described elsewhere [16]. We first define the syntax and the semantics of the subset MDG-HDL and Table language. We then define a set of functions, which translate the program from the MDG-HDL language to the Table languages. For each component in MDG-HDL, a compilation operator is defined as a function, which returns its Table code. A translation function *TransGT* is applied to each MDG-HDL program p so that the corresponding Table code is established. In other words,

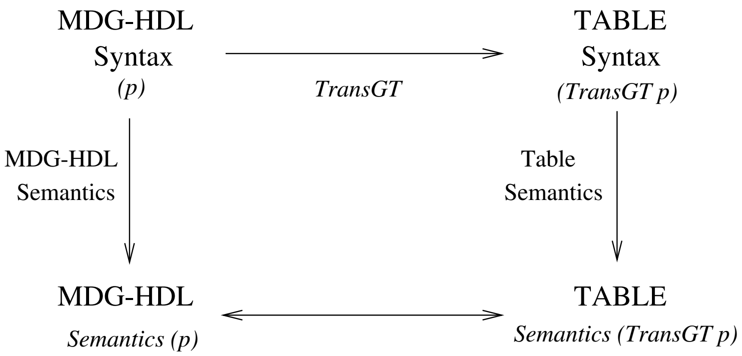

Fig. 3. Compilation Correctness

```
⊢ ∀  p. TransGT p = CorrespondingTablecode
```

The standard approach to proving a translator between two languages, is in terms of the semantics of the languages, as shown in Figure 3. Essentially the translation should preserve the semantics of the source language, which has the traditional form of compiler specification correctness used in the verification of a compiler [4]. The analogous method can be used to specify and verify the MDG system. For the translation to Table the correctness theorem has the form

```
⊢ ∀  p. Semantics (p) = Semantics (TransGT p)
```

4.1 MDG-HDL Syntax

In an MDG-HDL program, there is much information that is used in the MDG algorithm. When we write the syntax and semantics of programs, we can ignore this part of the information. Following the approach taken 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 into the form that we want.

For example, the MDG-HDL file of three *NOT* gates connected in series is given below.

```
...
signal(ip,bool).
signal(op,bool).
signal(u_B,bool).
signal(v_B,bool).
component(u_comp_B,not(input(ip),output(u_B))).
component(v_comp_B,not(input(u_B),output(v_B))).
component(op_comp_B,not(input(v_B),output(op))).
outputs([op]).
...
```

The abstract syntax of this file is

```
INTERNAL v_B (INTERNAL u_B (SEQ (NOT ip v_B
                                (SEQ (NOT v_B u_B) (NOT u_B op)))))
```

where *INTERNAL*, *SEQ* and *NOT* are syntactic constructors of the subset of the MDG-HDL language. More details will be given later.

The full abstract syntax of the subset of the MDG-HDL language is given in Figure 4. The MDG-HDL commands consist of predefined MDG-HDL components, an operation to set the initial value of a variable, a next state variable command, a composition operation and a localisation operation. The syntax of this language 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 35 constructors. The first 28 constructors are gates, flip-flops and registers. For example, the circuit term '*NOT ip op*' represents a *NOT* gate with one input labelled *ip* and one output labelled *op*.

The constructor *CONST1* declares a constant in a circuit. The constructor *FORK* represents the equality checker. 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 *SEQ* constructor represents the composition operation. If *c1* and *c2* are two values of type *mdg hdl*, then the term '*SEQ c1 c2*' represents the composition of the two terms represented by *c1* and *c2*. The *INTERNAL* constructor represents the localisation operation. If *c* is a term representing a circuit and *x* is a string (internal wire), then the circuit '*INTERNAL x c*' represents the circuit obtained by hiding the wire labelled *x* in the circuit represented by *c*.

The constructor *TABLESYN* represents the syntax of the MDG table component which has five arguments. The first argument is a list of inputs. The second argument 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:

```
out_type = NOWV of string|
           NEXTV of string
```

The third argument to a table is a list of table rows. Each row is a list itself, giving one allocation of values to the inputs. The entries in the list can be either actual values or a special don't care marker. This is realised by defining a new type (as given in [7]).

```
Table_Val = TABLE_VAL of α | DON'T_CARE

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 final argument is the default value, 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. Again we define a new HOL type *default_type* in terms of the type *out type*.

```
default_type = DENORMAL of num->bool|
               DEOUT of out_type|
```

For example, the syntax of a *NOT* gate table is given below:

```
TABLESYN [ip] (NOWV op) [[TABLE VAL F];
                         [TABLE VAL T]]
                        [TSIG;FSIG] (DENORMAL ARB)
```

where "ARB" is the predefined HOL term representing an arbitrary value of a given type. The syntax of the MDG-HDL program can be any *mdg_hdl* term.

```
program = PROG of mdg_hdl
```

4.2 Table Syntax

The MDG Table language is a subset of the MDG-HDL language. It only consists of five of the constructors that we mention above– *INIT, SNXT, TABLESYN, SEQ* and *INTERNAL*. We do not define a new type for the MDG Table language. However, when we translate the MDG-HDL program into the MDG Table program, the Table program only consists of those five constructors. For example, the Table code of the three *NOT* gates is

```
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 |
          CONST1 of bool=>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 |
          SEQ of mdg_hdl=>mdg_hdl |
          INTERNAL of string => mdg hdl

program = PROG of mdg hdl
```

Fig. 4. The Syntax of the MDG-HDL Program

```
INTERNAL v_B (INTERNAL u_B
 SEQ (TABLESYN [ip] (NOWV u_B) [[TABLE_VAL F];
                              [TABLE_VAL T]]
                             [TSIG;FSIG] (DENORMAL ARB)
      SEQ (TABLESYN [u_B] (NOWV v_B) [[TABLE_VAL F];
                                     [TABLE VAL T]]
                                    [TSIG;FSIG] (DENORMAL ARB)
           TABLESYN [v_B] (NOWV op) [[TABLE_VAL F];
                                    [TABLE VAL T]]
                                   [TSIG;FSIG] (DENORMAL ARB))))
```

4.3 The Semantics of the MDG-HDL Program

We have defined the syntax of the MDG-HDL language. In this section, we will show how to define the semantics of an MDG-HDL program. First of all, the semantics of the MDG-HDL program is in terms of environment [11]. 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 s is for state variables and signals. Its value is a history function and has a type :num$\rightarrow$bool that represents functions from time (natural number) to the value at that time.

We define a semantic function *SemMdghdl* for MDG-HDL programs. The first 28 components are mainly logic gates and flip-flops. Traditional hardware semantics can be given. The semantics of a component is then a relation between the input values and the output values. For example, the *NOT* gate can be expressed by

```
SEM_NOT ip op (s:string->num->bool) = ∀ t. (s op t) = ~(s ip t)

SemMdghdl (NOT ip op) s = SEM NOT ip op s
```

The semantics of *CONST1* represents a constant in a circuit which takes a constant *const* as its value. The output value does not change at any time.

```
SEM_CONST const op (s:string->num->bool) =
                        (∀ t. s op t = const)
```

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

```
SEM_FORK ip op (s:string->num->bool) = ∀ t. ((s op) t = (s ip) t)
```

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

```
SEM_INIT (y:string#bool) (s:string->num->bool) =
                              (s (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.

```
SEM_SNXT ny y (s:string->num->bool) = (∀ t. s ny (t+1) = s y t)
```

Sequencing is defined inductively in terms of the component commands. The semantics of *SEQ* is the conjunction of the corresponding semantics of each sub-command.

```
SemMdghdl (SEQ c1 c2) s =
                     ((SemMdghdl c1 s) ∧ (SemMdghdl c2 s))
```

The semantics of *INTERNAL* uses existential quantification to hide the local variable from the environment. It adds another entry to environment s. s is still the environment for the external wires. However, the extra entry for the new internal wire is first checked. This effectively hides the internal wires in circuit term c.

```
SemMdghdl (INTERNAL x c ) s =
     ∃ z. SemMdghdl c (λy.( if (y = x) then z else s y))
```

The semantics of *TABLESYN* follows the semantics of the table that was given by Curzon et al [7]. They firstly defined a predicate *Table_match* to check if the input values match the table values.

```
Table_match inputs [] t = T ∧
Table_match inputs (CONS v vs) t =
 (((HD (inputs) t) = TableVal_to_Val (v: α Table Val)) ∨
       (v = DON'T_CARE)) ∧
    (Table_match (TL inputs) vs t) )
```

The function table checks if there is a match on each row. If there is then the output has the corresponding value. Otherwise, the output equals the default value.

```
(table ip (op:num ->β) ([]: α Table_Val list) list) V_out default t =
         (op t = default t)) ∧
(table ip op (CONS v vs) V_out default t =
    ((Table_match ip v t) =>
        (op t = (HD V_out) t) |
        (table ip op vs (TL V_out) default t)))
```

The semantics of the *table* is

```
TABLE ip (op:num ->β) (V_outs:(α Table_Val list) list) V_out default =
        ∀ t. table ip op V_outs V out default t
```

The semantics of *TABLESYN* is defined in terms of the function *TABLE*

```
SemMdghdl (TABLESYN ip (op:out_type) y3 y4 y5)) s =
   TABLE (MAP s ip) (SEM_OUTVAR op s) y3 y4 (SEM_DEFAULTVAR y5 s)
```

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

```
SemMdghdl (TABLESYN [ip] (NOWV op) [[TABLE_VAL F];[TABLE_VAL T]]
           [TSIG;FSIG] (DENORMAL ARB)) s =
              TABLE (MAP s [ip]) (SEM_OUTVAR (NOWV op) s)
                    [[TABLE_VAL F];[TABLE_VAL T]]
                    [TSIG;FSIG] (SEM_DEFAULTVAR (DENORMAL ARB) s)
```

Finally, the semantics of a whole MDG-HDL program is expressed as a function *SemMdghdl* inside the logic:

```
(SemMdghdl (NOT ip op) s = SEM_NOT ip op s) ∧
......
(SemMdghdl (FORK ip op) s = SEM_FORK ip op s) ∧
(SemMdghdl (TABLESYN ip op y3 y4 y5) s =
         TABLE (MAP s ip) (SEM_OUTVAR op s) y3 y4
                    (SEM_DEFAULTVAR y5 s)) ∧
(SemMdghdl (SEQ code1 code2) s =
        ((SemMdghdl code1 s) ∧ (SemMdghdl code2 s))) ∧
(SemMdghdl (INTERNAL x code) s =
         ∃ z. SemMdghdl code (λy. (if (y = x) then z else s y)))
```

4.4 Compiling MDG-HDL into the Table Language

The first step in specifying a compiler for MDG-HDL is to define a set of functions for compiling the MDG-HDL program into the Table language. For each component in MDG-HDL, a compilation operator is defined as a set of functions that return its table code. For example, a *NOT* gate is compiled into

```
TRANS_NOT (ip:string) op =
       TABLESYN [ip] (NOWV op) [[TABLE VAL F];
                                [TABLE_VAL T]]
                               [TSIG;FSIG] (DENORMAL ARB)
```

For the MDG-HDL program, we define a function *TransGT* inductively over the syntactic structure and this function translates the MDG-HDL program into the equivalent Table language.

```
(TransGT (NOT ip op) = TRANS_NOT ip op) ∧
......
(TransGT (SEQ (code1:mdg_hdl) code2) =
              SEQ (TransGT code1) (TransGT code2)) ∧
(TransGT (INTERNAL x code) = INTERNAL x (TransGT code))
```

For example, the following theorem obtained by rewriting with the definitions illustrates the translation of the MDG-HDL program of three *NOT* gates discussed above

```
⊢ TransGT (INTERNAL v_B (INTERNAL u_B
   (SEQ (NOT ip v_B (SEQ (NOT v_B u_B) (NOT u B op)))) =
     INTERNAL v_B (INTERNAL u_B
       SEQ (TABLESYN [ip] (NOWV u_B) [[TABLE_VAL F];
                                      [TABLE_VAL T]]
                                     [TSIG;FSIG] (DENORMAL ARB)
            SEQ (TABLESYN [u_B] (NOWV v_B) [[TABLE VAL F];
                                             [TABLE VAL T]]
                                            [TSIG;FSIG] (DENORMAL ARB)
                 TABLESYN [v_B] (NOWV op) [[TABLE VAL F];
                                           [TABLE VAL T]]
                                          [TSIG;FSIG] (DENORMAL ARB))))
```

4.5 Compiler Correctness Theorem

To verify the correctness of a translator as we suggested in the beginning of this section, we have to obtain a theorem that quantifies over its syntactic structure stating that the semantics of the MDG-HDL program is equivalent to the semantics of the Table program used in MDG implementation. For our subset language, we have proved a theorem by using HOL:

```
⊢ ∀  p. SemMdghdl p s = SemMdghdl (TransGT p) s
```

where *p* represents any MDG-HDL program and *TransGT* is the function defined earlier which translates the MDG-HDL program to its Table code. *s* is the environment disscussed earlier for variables, respectively. The correctness theorem is proved by structural induction on the syntax domain of the MDG-HDL program.

5 Conclusions and Further work

In this paper, we prove the correctness of the translation phase of a decision diagram system (the MDG system) using a theorem proving system (the HOL system). We have defined the syntax of a subset of the MDG-HDL language and the Table language in higher-order logic. The semantic function is defined by structural induction over their syntactic structure. A set of functions that translate the syntax of an MDG-HDL program to the syntax of the Table language has been defined. The correctness theorem, which quantifies over its syntactic structure, has been verified. This theorem states that the semantics of the original MDG-HDL program is equivalent to the semantics of the Table program used in the MDG implementation.

Our motivation for deep embedding the MDG-HDL and Table languages into HOL is not only to verify aspects of correctness of the MDG system, but also to make use of the semantics to formally import the MDG results into HOL based on a trusted MDG system (Figure 2). We have formally imported the correctness results produced by four different hardware verification applications into HOL [16]. 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 applications considered include combinational verification, sequential verification and invariant checking. Therefore, we can obtain theorems that justify the conversion of low level results proved in the MDG system to results about circuits in high level languages in a form that can be reasoned about in the HOL system.

The work presented in this paper is part of a larger project to verify a combined HOL-MDG system. We need to prove that the translation from the tabular code to MDG decision graphs is correct. We also need to prove that the MDG algorithms are correct. We need to extend the subset considered to deal with, for example, sort declaration for the verified system to be applicable to real designs.

Acknowledgments

We are grateful to Dr. Richard Boulton at University of Glasgow and Dr. Skander Kort at Concordia University for their help. This work is funded by EPSRC grant GR/M45221, and a studentship from the School of Computing Science, Middlesex University. Travel funding was provided by the British Council, Canada.

References

1. 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.
2. R. S. Boyer and G. Dowek. Towards checking proof checkers. In *Workshop on Types for Proofs and Programs (Type'93)*, 1993.
3. R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions in Computers*, 35(8):677–691, August 1986.
4. 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.
5. 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.
6. 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.
7. 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.
8. 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.
9. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-order Logic.* Cambridge University Press, 1993.
10. P. V. Homeier and D. F. Martin. A verified verification condition generator. *The Computer Journal*, 38(2):131–141, July 1995.

11. T. F. Melham. *Higher Order Logic and Hardware Verification*. Cambridge Tracts in Theoretical Computer Science 31. Cambridge University Press, 1993.
12. J. von Wright. Program refinement by theorem prover. In *Proc. 6th Refinement Workshop*, London, January 1994. Springer–Verlag.
13. J. von Wright. Representing higher-order logic proofs in HOL. *The Computer Journal*, 38(2):171–179, July 1995.
14. J. von Wright. The formal verification of a proof checker. SRI internal report, November 1998.
15. W. Wong. Validation of HOL proofs by proof checking. *Formal Methods in System Design*, 14(2):193–212, March 1999.
16. H. Xiong, P. Curzon, and S. Tahar. Importing MDG verification results into HOL. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics*, number 1690 in Lecture Notes in Computer Science, pages 293–310. Springer-Verlag, September 1999.
17. Z. Zhou and N. Boulerice. *MDG Tools (V1.0) User Manual*. University of Montreal, Dept. D'IRO, 1996.