Proving Existential Theorems when Importing Results from MDG to 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. An existential theorem, for the specification or implementation of hardware, states that for any inputs there must exist at least one output which is consistent with it. It is proved to prevent an inconsistent model being produced and it is required to formally import the verification result from one verification system to another system. In this paper, we investigate the verification of the existential theorems of hardware specifications and implementations. Whilst much of the approach is generally applicable, we specifically consider a hybrid system linking the MDG hardware verification system with the HOL interactive proof system. 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.

1 Introduction

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 hybrid tools have been developed such as the Hol-Voss system [8], Forte [1], HOL-MDG [9] etc. Normally, the verification results from one system are translated to another system. In other words, there is a linkage between the two systems. How can we ensure that this linkage can be trusted?

Many different technologies have been used to link two different systems in a trusted way. Gordon [6] integrated the BDD based verification system BuDDY into HOL by implementing BDD-based verification algorithms inside HOL building on top of primitives provided. Since "LCF-Style" general infrastructure was 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 it also made it possible to achieve the advantages of both theorem proving tools and state algorithms. Hurd [7] used a different method to combine the strengths of two theorem-prover systems– Gandalf and HOL. He wrote functions to simulate the Gandalf proof according to the Gandalf logged file so reconstructing 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.

The HOL-MDG hybrid system uses another way to make the linkage more natural and trustworthy. The MDG system is a symbolic state enumeration system based on Multiway Decision Graphs (MDGs) [4]. The linkage between the two systems is based on a series of importing theorems [11], which formally convert the formalized MDG verification results in a form usable in a traditional HOL hardware verification, i.e., the structural specification implements the behavioral specification. The 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. The importing theorem for the sequential verification has the form:

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

where `SPEC` represents the behavioral specification and `IMPL` represents the structural specification. The theorem we require in HOL is:

$$\vdash_{thm} \forall\ \texttt{ip op}.\ (\texttt{IMPL ip op} \supset \texttt{SPEC ip op}) \tag{1}$$

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 the 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):

$$\vdash_{thm} \forall\ \texttt{ip}.\ \exists\ \texttt{op}.\ \texttt{SPEC ip op} \tag{2}$$

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 (1). This representation might meet an inconsistent model that trivially satisfies any specification. This is sometimes called "The false implies anything problem" [3]. If the implementation of a design (`IMPL ip op`) 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 specification (`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 [10], which has the form:

$$\vdash_{thm} \forall\ \texttt{ip}.\ \exists\ \texttt{op}.\ \texttt{IMPL ip op} \tag{3}$$

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 paper, 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 [12]. 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 paper, 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:

$$\vdash_{thm} \forall \texttt{ ip. } \exists \texttt{ op. C ip op} \tag{4}$$

For example, if we consider a circuit consisting of two NOT gates in series, the existential theorem for this circuit should be (SEM_NOT ip op represents the semantics of the NOT gate):

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

In fact, the stronger consistency theorem (3) is an `existential theorem` for the structural specification, whereas the additional assumption (2) 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 [?], which is used to eliminate existentially quantified variables in a goal. This tactic corresponds to a theorem EXISTS_ELIM given below.

$$\vdash_{thm} \texttt{(}\exists\texttt{x. (x = t) } \wedge \texttt{ (A x)) = A t} \tag{5}$$

In other words, if the existentially quantified variable (`x`) is explicitly represented by its value as in (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 paper, we concentrate on proving the existential theorems based on the syntax and semantics of MDG-HDL [12] [5]. 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`.

The structure of this paper is as follows: in Section 2, we overview the MDG system. We will briefly introduce the semantics and syntax of the MDG input language (MDG-HDL) in Section 3. In section 4, we not only give the detail about proving the existential theorems for each component in the MDG library, but also we provide a method which is used to prove the existential theorem for any specification and implementation of a design. Finally, conclusions are presented in Section 5.

2 The MDG system

The MDG system is a hardware verification system based on Multiway Decision Graphs (MDGs). MDGs subsume the class of Bryant's Reduced Ordered Binary Decision Diagrams (ROBDD) [2] while accommodating abstract sorts and uninterpreted function symbols. The system combines a variety of different hardware verification applications implemented using MDGs [13]. The applications developed include: combinational verification, sequential verification, invariant checking and model checking.

The input language of MDG is MDG-HDL [13], which is a Prolog-style hardware description language and 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 constructor 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.

3 The Syntax and the Semantics of MDG-HDL

The `existential theorems` we consider are theorems about the meaning of MDG-HDL programs. We need to define first the syntax and semantics of the language. The abstract syntax of an MDG-HDL program is in terms of the MDG circuit description file, which consists of an external output wires list, an external input wires list, an internal wires list and a component term. A component term describes how circuits are constructed from subcircuits with the exception of hiding operation on internal wires. 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.

In the MDG-HDL language, the inputs and outputs of the component `TABLE` could be different sorts. These sorts could be boolean sorts, concrete sorts or abstract sorts. In this paper, we will not consider the abstract sorts. We consider a subset language of MDG-HDL whose inputs and outputs of a `table` could be boolean sorts or 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. We define a new type `Mdg_Basic` in HOL to meet this requirement.

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

A semantic function `SemProgram` is defined for giving the semantics of the MDG-HDL programs, which is in terms of the semantics of the MDG-HDL component term abstract syntax (`SemMdghdl`) and a hiding operation of the internal wires. Since the type of inputs and outputs of the component is `Mdg_Basic` above, we need to judge if the inputs and outputs are of boolean value by predicate `IS_BOOL`. The semantics of the logic gates and flip-flops are then a conjunction of the judgments 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 justify if the value of an `Mdg_Basic` term is `BOOL T` or `BOOL F`, and function `MDG_TO_BOOL` converts the `Mdg_Basic` term `BOOL T` and `BOOL F` to boolean value `T` and `F`.

An MDG Table component is essentially a flexible form of Truth Table, giving the value of an output in terms of variable values. For example, the table specifying a NOT gate can be formalized as

```
⊢def NOT_TABLE x y =
      TABLE [x] y [[TABLE_VAL (BOOL T)];
                   [TABLE_VAL (BOOL F)]] [FSIG1;TSIG1] (λt. ARB)
```

The first argument to TABLE is a list of input terms, the second the variable being defined, the third gives possible values for the inputs with the fourth giving corresponding values on the outputs. The final argument gives the value of the output in any cases not specified. Thus the above table states that if `x` is `true` then the output `y` is `false`, if `x` is `false` then the output `y` is `true`, otherwise the output `y` is an arbitrary value.

The semantics of the table construct was initially given by Curzon et al [5]. 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 may have unequal lengths, when either list is empty, the output value equals the default value.

```
⊢def (table ip (op:num ->β) [] V_out default t = (op t = default t)) ∧
     (table inps out vs [] default t = (out 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 times.

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

where functions `Justify_Type` and `Justify` are used to check the type of each input and output of a `table`.

The semantics of the MDG-HDL component term is represented as a function `SemMdghdl` inside the logic:

```
⊢def (SemMdghdl (NOT ip op) env = SEM_NOT (env ip) (env op)) ∧
     ......
     (SemMdghdl (JOIN code1 code2) env =
             ((SemMdghdl code1 env) ∧ (SemMdghdl code2 env)))
```

where `env` is an environment, which is a function that has type :string $\rightarrow\delta$. This function maps a variable name (modeled by strings) to the value of that variable. The hiding operation 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 =
        (∃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, 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→Mdg_Basic)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 = env) ∧
     (Dsem_Ext (CONS (v:string) vs) env var =
          (Dsem_Ext vs (λwv.if (wv = v) then (HD var)
                                         else (env wv)) (TL var)))
```

Finally, the semantics of a program is described by `SemProgram`, which explicitly represents the relation between the external inputs and outputs. 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.

```
⊢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
               (Justify_Condition code env2 (SemInvariable inv))⊃
                      Dsem_Int (SemInvariable inv) code env2
```

where functions `SemExoutput`, `SemExinput` and `SemInvariable` are defined to access values of the external output and input wires and internal wires, function `Justify_Condition` is defined to check whether each external wire has boolean type or concrete type.

For example, the syntax a `NOT` gate circuit could be expressed as:

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

The corresponding semantics of it is:

```
(∀t. IS_BOOL (HD ip t) ∧ IS_BOOL (HD op t)) ⊃
                    SEM_NOT (HD ip t) (HD op t)
```

where first line of the semantics is the results of checking the type of each external inputs and outputs by using the function `Justify_Condition`. It states that if the external wires have proper types then the semantics of the program should be the semantics of circuit (`Dsem_Int`). In this example the semantics of the circuit is `SEM_NOT`. By expanding the sematics of the `NOT` gate and simplifying it, we obtain the semantics of the MDG-HDL program (one `NOT` gate) is:

```
∃ 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)
```

4 Solving Existential Goals

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 rewriting 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 section, 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 existential quantified internal variable. Finally, we give an example demonstrates how to apply our approach to prove the `existential theorem` of a whole circuit.

4.1 The Output Representation for the Basic MDG-HDL Components

In the MDG-HDL library, there are two classes of non-table component. One is that the output of a component is a signal variable (ie non state holding, the other is that the output of a component is a state variable. The `existential term` for the two classes is 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

```
⊢def existnot (ip:Mdg_Basic) =
        (Bool1_Mdg ∼ (λwv. (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, the 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`.

4.2 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:(num->α )list) [] u_out (default:num->β) t =
                              (default:num->β) t) ∧
     (existtable (input:(num->α)list) vs [] (default:num->β) t =
                              (default:num->β) t) ∧
     (existtable input (CONS v vs (CONS u u_out) default t =
                     (if (Table_match input (v:α Table_Val list) 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 the 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.

```
∀ 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
```

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 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
```

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 the `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 the `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`.

4.3 Dealing with the Existential Quantified Internal Variables

When we prove the existential theorem for a circuit, if the circuit contains internal wires, then we also need to strip away these wires. The `existential term` for these wires are nearly the same as we mentioned above. A difference is that the type of these wires is `:num -> Mdg_Basic` rather than `:(num -> Mdg_Basic) list`.

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))
```

where `x1` is an internal wire who 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 prove by using tactic `PROVE_EXIST_TAC`.

4.4 Example

Example 5. Consider the circuit shown in Figure 1. 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.

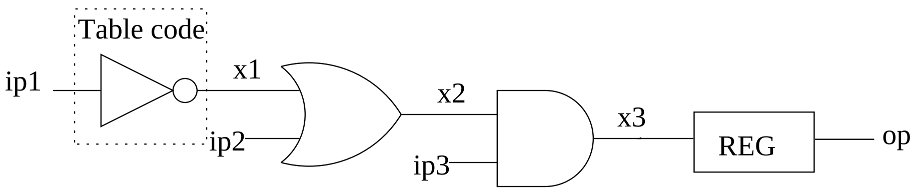

Fig. 1. Example

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 finally can 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 a 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.

5 Conclusions and Further work

In this paper, we summarized a general way of how to prove the existential theorem for the implementation or specification of hardware 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 is used to strip away the leading existentially quantified variable and substitutes `term` for each free occurrence in the body, is determined in terms of those `output representations`. The proving process normally can be considered as three steps. Since we directly deal with the syntax and semantics of the MDG-HDL program, we first use a tactic `EXPAND_SEMANTICS_TAC` to expand the semantics of the program (design) and obtain a HOL goal of the form ∃ `a1 ... an. body`. We then use the `existential term` to strip away the existentially quantified variable and substitute `term` for each free occurrence in the body. However, doing such existential elimination still needs user guidance. Finally, a further two tactics `PROVE_EXIST_TAC` and `PROVE_TABLE_EXIST_TAC` are used to solve the resulting goal.

The reason that we prove the `existential theorem` is to easily import the MDG results into HOL and avoid an inconsistent model occurring. We intend to further refine the tactics provided so that the existential theorem can be proved more efficiently and easily. This would remove the burden from the user of the MDG-HOL hybrid system.

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 paper, our methods can be explored to slove other HOL goals which are existentially quantified, since it also can be viewed as an `existential theorem`. In fact, we also provide a library for giving the `output representation` of each component in a boolean subset. It can be used to construct the `existential term` in the HOL goal. In other words, our `existential terms` and `output representations` could be used to solve some existential quantified HOL goals in the other applications.

Acknowledgments

We are grateful to Dr. Konrad Slind and Dr. Michael Norrish at University of Cambridge 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. 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.

2. R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computer Surveys*, 24(3), September 1992.
3. 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.
4. 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.
5. 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.
6. M. J. C. Gordon. Reachability programming in HOL98 using BDDs. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, number 1869 in Lecture Notes in Computer Science, pages 179–196. Springer-Verlag, August 2000.
7. J. Hurd. Integrating GANDALF and HOL. Technical Report 461, University of Cambridge, Computer Laboratory, April 1999.
8. J. Joyce and C. Seger. Linking BDD-based symbolic evaluation to interactive theorem-proving. In *the 30th Design Automation Conference*, 1993.
9. S. Kort, S. Tahar, and P. Curzon. Hierarchical verification using an MDG-HOL hybrid tool. In *IFIP Conference on Correct Hardware Design and Verification Methods (CHARME'2001)*, Livingston, Scotland, UK, September 2001.
10. T. F. Melham. *Higher Order Logic and Hardware Verification*. Cambridge Tracts in Theoretical Computer Science 31. Cambridge University Press, 1993.
11. 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.
12. H. Xiong, P. Curzon, S. Tahar, and A. Blandford. Embedding and verification of an MDG-HDL translator in HOL. In *TPHOLs 2000 Supplemental Proceedings*, pages 237–248, August.
13. Z. Zhou and N. Boulerice. *MDG Tools (V1.0) User Manual*. University of Montreal, Dept. D'IRO, 1996.