Providing a Formal Linkage between the MDG Verification System and HOL Proof System

Haiyan Xiong, Paul Curzon,
Sofiène Tahar and Ann Blandford

Technical Report:IDC-TR-2003-006
December 2003

Interaction Design Centre

Interaction Design Centre
School of Computing Science
Middlesex University
Trent Park Campus
Bramley Road
N14 4YZ

For further details of this technical report series contact:
Paul Curzon (p.curzon@mdx.ac.uk)

Providing a Formal Linkage between MDG and HOL

Haiyan Xiong[1], Paul Curzon[1], Sofiène Tahar[2], and Ann Blandford[3]

[1] School of Computing Science, Middlesex University, London, UK.
Tel: +44 20 84116762, Fax: 44 20 84115924
p.curzon@mdx.ac.uk
[2] ECE Department, Concordia University, Montreal, Canada.
tahar@ece.concordia.ca
[3] UCL Interaction Centre, University College London, London, UK.
a.blandford@ucl.ac.uk

Abstract. We describe an approach for formally linking a symbolic state enumeration system and a theorem proving system based on a verified version of the former. It has been realized using the HOL system and a simplified version of the MDG system. It involves the following three steps. Firstly, we have verified aspects of correctness of a simplified version 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 a set of theorems, which formally import MDG verification results into HOL theorems. Thirdly, we have combined the translator correctness and importation theorems to allow MDG verification results to be imported in terms of a high level language (*MDG-HDL*) rather than low level decision diagrams. We also summarize a general method of the stronger consistency theorem to prove design implementations against respective specifications. The feasibility of this approach is demonstrated in a case study that integrates two applications: hardware verification (in MDG) and usability verification (in HOL). A single HOL theorem is proved that integrates the two results.

1 Introduction

Deductive theorem proving and symbolic state enumeration are complementary approaches to formal verification. In the former, 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) [3].

Deductive theorem proving systems often use interactive proof methods. The user interactively constructs a formal proof which proves a theorem stating the correctness of an implementation. Theorem proving systems allow a hierarchical verification method to be used to model the overall functionality of designs with complex datapaths. They are very general in their application. Theorems cannot 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 and modeling and verifying a system is very time-consuming. This is a major problem for applying theorem proving systems in industry.

In contrast, symbolic state enumeration systems are automated decision diagram based 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 symbolic state enumeration verification method 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.

Recently, there has been a great deal of work concerned with combining theorem proving and symbolic state enumeration systems to gain the advantages of both. A common approach to combining proof tools is to use a symbolic state enumeration system as an oracle to provide results to the theorem proving system. In other words, an oracle is used to receive problems and return answers. For example, the HOL system [17] provides approaches for tagging theorems that are dependent on the correctness of external verification tools. An oracle can be built in the HOL system and viewed as a plug-in. The issue in such work is to guarantee that the results provided by external tools are theorems within the theory of the proof system. This process thus brings about two questions:

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

We investigate the answers to the above two questions. Some symbolic state enumeration based systems such as MDG [8] 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 to the first question. To solve the second question, we can formally justify the correctness results obtained from the symbolic state enumeration system in a theorem prover.

The main contribution of this paper is that we describe an approach that provides 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 partly realize the methodology with the HOL system and a simplified version of the MDG system. We prove the correctness of aspects of the simplified MDG system. We also provide a formal linkage between the HOL system and the simplified MDG system based on the importing of MDG results to HOL theorems [31]. Most importantly, we combine the translator correctness theorems with the importation theorems in order to allow low level MDG verification results to be imported into HOL in terms of the semantics of a high level language (*MDG-HDL*). Full details of this work can be found in [30]. Lessons from the work are applicable to other related systems. We chose HOL and MDG because this work is part of a large project in collaboration with the Hardware Verification Group at Concordia University. They are developing a hybrid system (MDG-HOL) [24] which combines the MDG system and the HOL system.

The structure of the rest of this paper is as follows: in Section 2, we review related work. In Sections 3 and 4, we briefly introduce the MDG and HOL systems, respectively. The main body of the paper is described in Section 5. Section 6 presents a case study application of our work. Finally, our conclusion and ideas for further work are presented in Section 7.

2 Related Work

Many different technologies have been used to link verification systems. We will briefly review a range of work linking proof systems to external, automated verification tools to give a flavour of the approaches. We concentrate on higher-order logic proof systems since that is the main focus of this paper.

Joyce and Seger [23] presented a hybrid verification system, HOL-Voss, which links HOL to a symbolic trajectory system. Predicates were defined in the HOL system, which created a mathematical link between the specification language of the Voss system (symbolic trajectory evaluation) [20] and that of the HOL system. Aagaard *et al.* built on this work in developing the Forte verification system [1]. Forte is a combined model checking (in Voss via symbolic trajectory evaluation) and theorem proving system (ThmTac). ThmTac is written in `fl` (a strongly-typed functional language in the ML family [27]) and is an LCF (Logic of Computable Functions) style implementation of a higher-order classical logic. Both specification and implementation language of Forte 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.

Rajan *et al.* [28] proposed an approach for the integration of model checking with PVS [9]: the Prototype Verification System. The μ-calculus, which consists of quantified Boolean formula and predicates defined by means of the least and greatest fixpoint operators, was used as a medium for communicating between PVS and a model checker.

Computer algebra tools are another class of tool that can provide useful automated results for a theorem prover. Harrison and Théry [19], for example, combined the theorem prover system (HOL) and the Maple [21] computer algebra system. A software bus with three different processes: HOL, Maple, and a bridge were used to connect the theorem prover and computer algebra system. A request is sent by HOL which is received and translated by the bridge, and then sent to the computer algebra system. The answer from the computer algebra system is then transferred back to the prover through the bridge.

In work such as the above an external tool is trusted to provide results to the proof system. The external system is assumed to produce correct results (at least with a similar level of confidence as the proof system). Gordon [16] investigated a way that such trust can be increased. He integrated the BDD based verification system BuDDY [25] 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 those primitives are correct — not only could the standard state enumeration 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 enumeration algorithms, without the need to trust a complete external package, just a set of primitives.

Hurd [22] used a different method to combine the strengths of two systems without loss of trust. He considered two theorem-prover systems—Gandalf [29] 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. Gandalf thus finds a proof externally but that proof is then actually recreated in HOL. As a result, the Gandalf proof results need not be tagged into HOL and the degree of trust is high. If the Gandalf proof is incorrect, HOL will just fail to create a theorem when following it.

In the above cases the effort was to link a single tool to a proof system. Other work has looked at providing more general infrastructure for linking a variety of tools. For example, the PROSPER toolkit [14] provides a uniform way of linking HOL with external proof tools. The specification of its integration interface has been implemented in several languages allowing components written in these languages to be used together. A range of different external proof tools can access the toolkit and act as servers to a HOL client. It also tags theorems produced by its plug-in with a label which can be used in the HOL system. The MDG-HOL system [24] used the PROSPER/Harness Plug-in Interface to link the HOL system and the MDG system.

The VeriTech project [18] developed an interactive tool to integrate a variety of formal verification tools together. It is based on a set of tools translating from each component verification tool to a core representation and from the core to each tool. This translation allows the user to directly import one verification tool's specification and implementation files into another verification tool. The user can thus take advantage of the different verification tools.

We take a different approach from the above work. The external system is not trusted unreservedly as in the earlier linkage work. Instead the proof system is used to verify aspects of its correctness. Furthermore the linkage is not trusted implicitly either – linkage theorems verify the way results are turned into theorems in the proof system. We focus on the verification of a symbolic state enumeration system (the MDG system) and provide a theoretical underpinning to the formal linkage of such a system and a theorem proving system (MDG and HOL). We verify the correctness of translators of the MDG system by using the HOL system and prove theorems that formally convert the MDG verification results of MDG's different applications into the traditional HOL hardware verification theorems. By combining the translator correctness theorems with the importation theorems, the MDG verification results can be imported into HOL in terms of the MDG input language (*MDG-HDL*). The feasibility of this approach is demonstrated in a case study that integrates two distinct applications: hardware verification (in MDG) and usability verification (in HOL). A single HOL theorem is proved that integrates the two results.

3 The MDG System

The full MDG system [8] 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) [4] 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 $\vee_{1 \leq i \leq n} \; ((A = B_i) \wedge P_i)$.

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 rather than a vector of Boolean variables, 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. 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 [34] 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.

The input language of the MDG system is a Prolog-style hardware description language (*MDG-HDL*) [34], which supports structural specification, behavioral specification or a mixture of both. A structural specification is usually a netlist of components connected by signals, and a behavioral specification is given by a tabular representation of transition/output relations or a truth table.

4 The HOL System

HOL [17] is a theorem proving environment, which uses higher-order logic to model and verify systems. There are two main proof methods used: 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. It is an LCF [15] style proof system: 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 [27], called *tactics* and *tacticals*, are applied that break a proof goal into a list of subgoals. Tactics and tacticals are repeatedly applied to the subgoals until they can be proved. 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 a goal's assumptions to a suitable form. Table 1 shows the notation of higher-order logic and corresponding meaning used in this paper.

Theorems in the HOL system are represented by values of the SML abstract type `thm`. In a pure system (where `mk_thm` which creates arbitrary theorems is not used), a theorem is only obtained 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 job. In this way, the SML type system protects the HOL logic from the arbitrary construction of a theorem, so that every computed value of the type representing theorems is a theorem. The user can have a great deal of confidence in the results of the system provided `mk_thm` and new axioms are not used.

Notation	Meaning
T	truth
F	falsity
P(x)	x has property P
t1 $\wedge$ t2	t1 and t2
t1 $\vee$ t2	t1 or t2
t1 $\supset$ t2	t1 implies t2
$\forall$ x. t[x]	for all x, it is the case that t[x]
$\exists$ x. t[x]	for some x, it is the case that t[x]

Table 1. Higher-order Logic Notation

HOL has a rudimentary library facility which enables 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.

5 Formally Linking a Verified MDG and HOL System

The intention of our work 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 in terms of the MDG input language as shown in Figure 1. This work can be divided into three steps.

- We must verify the correctness of the MDG system using the HOL system. It consists of two phases: (1) verification of the translators (step 1 of Figure 1) [32] and (2) verification of the algorithms (step 2 of Figure 1).
- We then must prove theorems (step 3 of Figure 1), which formally convert the verification results of the MDG applications into traditional HOL hardware verification theorems [31].
- By combining the correctness theorems (theorems obtained from step 1 of Figure 1 and step 2 of Figure 1) of the verification of the MDG system with the importing theorems (obtained from step 3 of Figure 1), the MDG verification results can be formalized in terms of the MDG input language (*MDG-HDL*) in a form suitable for use in HOL.

During this study, we concentrate on the verification of the translation phase of the MDG system (step 1 in Figure 1) using the HOL theorem prover and importing the MDG results into HOL to form the HOL theorems (step 3 in Figure 1) [31]. Step 2 is similar to Chou and Peled's work [7] which verifies a partial-order reduction technique for model checking. Verifying the algorithms is beyond the scope of this paper: we are primarily concerned with the linkage and how it could be combined with the correctness theorems and importation theorems. We outline the overall methodology and emphasize the importation process

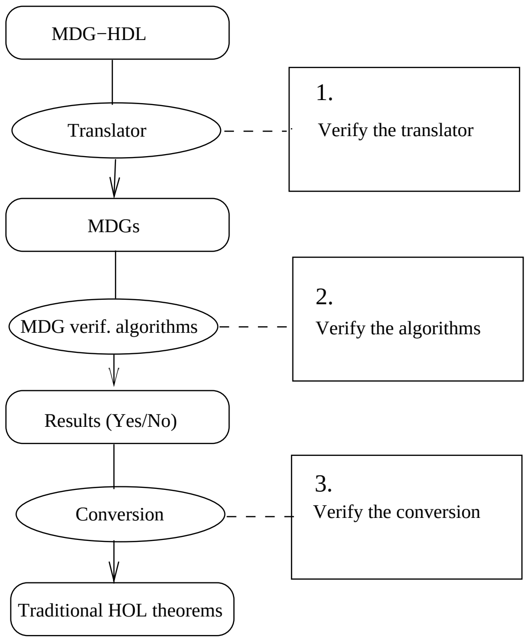

Fig. 1. Overview of the Formal Linkage Project

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 HOL theorems based on the semantics of the high level MDG input language (*MDG-HDL*) [34] rather than the semantics of the low level MDG results. Since we use a deep embedding semantics, the separate translator correctness theorems can be combined with each other and the importation theorems. Besides, we also summarize a general method for proving stronger consistency theorems, which occur as assumptions to the importation theorems, to remove the burden from the user of the combined system [32]. These theorems are needed specifically to justify importing sequential verification results on sequential designs into the theorem proving system.

In the remainder of this section, we will discuss the individual steps that we have undertaken: verifying the translator correctness theorems, proving the general importation theorems, combining the translator correctness theorems with the importation theorems in terms of deep embedding semantics, proving the *existential theorem* and implementing our method in a case study that integrates two different applications.

5.1 Verifying the MDG Translators

The input language of the MDG system (*MDG-HDL*) supports structural specification, behavioral specification or a mixture of both. The various specifications

Table([[x1, x2, y], [1, 1, 1] | 0)

INPUTS		OUTPUTS
x1	x2	y
T	T	T
		F

Fig. 2. The AND Table

$$\begin{tikzcd}
\text{MDG-HDL} \arrow[r, "(1)"] & \text{core MDG-HDL} \arrow[r, "(2)"] & \text{MDGs}
\end{tikzcd}$$

Fig. 3. Overview of the MDG Translation Phases

of a design are input as a series of files. In particular, 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 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 a `table` as shown in Figure 2. It states that if `x1` is equal to `true` and `x2` is `true` then the output `y` is equal to `true`, otherwise the output `y` is equal to `false`.

In the MDG system, most of the components in the MDG-HDL library are compiled into their own *core MDG-HDL* tabular code first. The *core MDG-HDL* program can then be compiled into an internal *MDG*. Some components, such as registers, are implemented directly in terms of an *MDG*. However, in theory these components could also be implemented as tables; this provides a general specification mechanism. We assume the *MDG-HDL* program is firstly translated into a *core MDG-HDL* program. The *core MDG-HDL* program is then translated into *MDGs*. In this situation, the MDG system could be specified as in Figure 3.

Adopting this approach makes the translation phase more amenable to verification. We are not verifying the actual MDG implementation (although an MDG system could be implemented in this way, the existing implementation is not). Rather, our formalization of the translator is a specification of it. Once combined with a translator from *core MDG-HDL* to *MDG*s, it would be specifying the output required from the implementation. This would be used as the basis for verifying such an implementation. We thus 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 and specification correctness was advocated by Chirica and Martin [6] with respect to compiler correctness.

We define a deep embedding semantics for a subset of the *MDG-HDL* language. This subset is all of MDG except three predefined components, namely the Multiplexer, the Driver and the Transform construct used to apply functions. These components are omitted from our subset as they have non-Boolean inputs or outputs. We consider this subset since our aim is to explore the feasibility of this method. The subset does allow a program to contain concrete sorts. In other words, the inputs and outputs of a `table` could be Boolean 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. To cope 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. In the rest of this paper, we will refer to this simplified version of the MDG system as "the MDG system".

We verified the first translation step of the MDG system (see phase 1 in Figure 3) based on the syntax and semantics of the MDG input language (*MDG-HDL*) and the *core MDG-HDL* language using the HOL theorem prover. The syntax and the semantics of the subset *MDG-HDL* and *core MDG-HDL* are defined. A set of functions, which translate the program from *MDG-HDL* to *core MDG-HDL* is then defined. For each program in *MDG-HDL*, the compilation operators are defined as functions, which return their *core MDG-HDL* code. A translation function `TransProgMC` is applied to each *MDG-HDL* program so that the corresponding *core MDG-HDL* program is established (Figure 4). In other words, the relations of the translations can be represented as follows:

$\forall$ `p. TransProgMC p =` *Corresponding core MDG-HDL program*

The standard approach to prove a translator between two languages is in terms of the semantics of the languages. Essentially the translation should preserve the semantics of the source language. This has the traditional form of compiler specification correctness (Figure 4) used in the verification of a compiler [6]. An analogous method has been used in the specification and verification of the MDG system. For the translation to *core MDG-HDL*, the correctness theorem has been proved:

$$\vdash_{thm} \forall\ \texttt{p. SemProgram (p) = SemProgram\_Core (TransProgMC p)} \qquad (1)$$

where `SemProgram` and `SemProgram_Core` are semantic functions for the *MDG-HDL* program and *core MDG-HDL* program. This theorem states that the semantics of the low level *core MDG-HDL* program is equal to the semantics of

$$\begin{tikzcd}
\begin{array}{c}\text{MDG−HDL}\\\text{Syntax}\\ \textit{p}\end{array} \arrow[r, "\textit{TransProgMC}"'] \arrow[d, "\begin{array}{c}\text{MDG−HDL}\\\text{Semantics}\end{array}"'] & \begin{array}{c}\text{core MDG−HDL}\\\text{Syntax}\\ \textit{TransProgMC p}\end{array} \arrow[d, "\begin{array}{c}\text{core MDG−HDL}\\\text{Semantics}\end{array}"'] \\
\begin{array}{c}\text{MDG−HDL}\\ \textit{SemProgram (p)}\end{array} \arrow[r, leftrightarrow] & \begin{array}{c}\text{core MDG−HDL}\\ \textit{SemProgram\_Core (TransProgMC p)}\end{array}
\end{tikzcd}$$

Fig. 4. Compilation Correctness

the high level *MDG-HDL* (the MDG input language). For a subset of MDG, we have also verified the translation to a lower level formula language of decision diagrams. More details can be found in [30].

5.2 The Importation Theorems

Generally, when we use HOL to verify a design, the design is modeled as a hierarchy structure with modules divided into submodules as shown in Figure 5. The submodules are repeatedly subdivided until the logic gate level is eventually 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:

$$\texttt{implementation} \supset \texttt{specification} \tag{2}$$

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 formalized 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 results does not give theorems of the form that HOL needs if we are to use traditional HOL hardware verification techniques. Therefore, we have

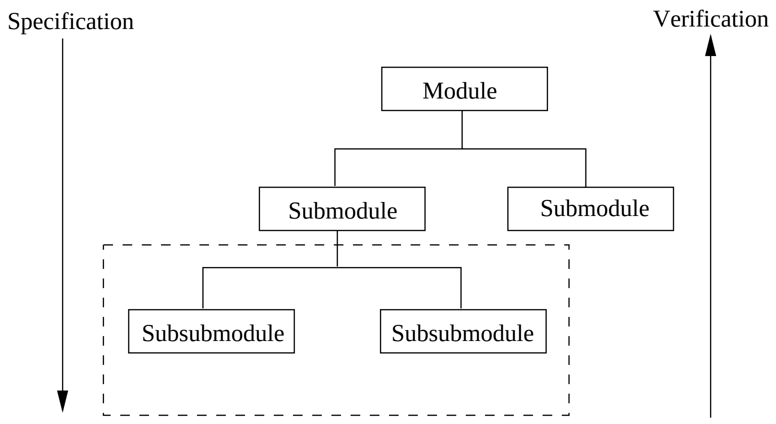

Fig. 5. Hierarchical Verification

to convert the MDG results into a form that can be used. In other words, we proved a series of translation theorems (one for combinational verification, one for sequential verification, etc.) that state how an MDG result can be converted into the traditional HOL form:

$$\texttt{Formalized\_MDG\_result} \supset \texttt{implementation} \supset \texttt{specification} \tag{3}$$

We have formally specified the correctness results produced by several different MDG verification applications and given general *importation theorems*. 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 applied to other verification systems with similar architectures based on reachability analysis or equivalence checking.

For example, the behavioral equivalence of two state machines (Figure 6) is verified by checking that the machines produce the same sequence of outputs for every 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 6) 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. The result that MDG proves about `PSEQ` is that the `flag` output is always true. This can be formalized as

$$\forall\ \texttt{ip op op' flag. PSEQ ip op op' flag M M'} \supset (\forall\ \texttt{t. flag t = T}) \tag{4}$$

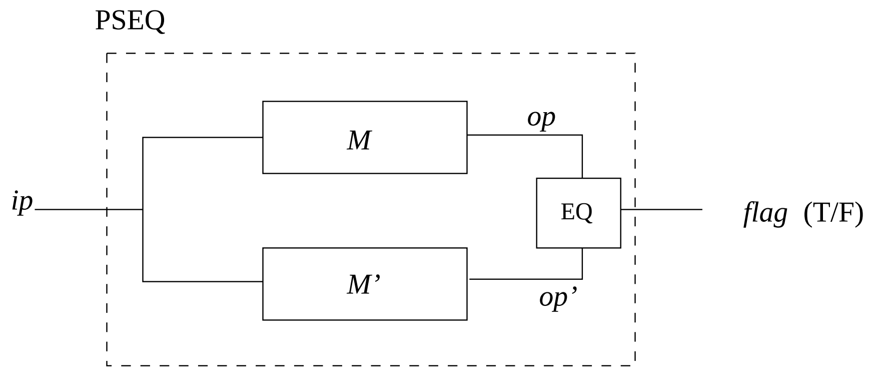

Fig. 6. The Product Machine used in MDG Sequential Equivalence Checking

The corresponding importation theorem which converts MDG results to the appropriate HOL form has been obtained:

$$\begin{aligned}
&\vdash_{thm} \forall \texttt{M M'}.\\
&\qquad((\forall \texttt{ ip op op' flag}.\\
&\qquad\quad \texttt{PSEQ ip op op' flag M M'} \supset (\forall \texttt{ t. flag t = T})) \wedge\\
&\qquad(\forall \texttt{ ip}. \exists \texttt{ op'. M' ip op'})) \supset\\
&\qquad\qquad(\forall \texttt{ ip op. M ip op} \supset \texttt{M' ip op})
\end{aligned} \tag{5}$$

This suggests that the MDG results can only safely be imported into HOL when an additional assumption ($\forall$ `ip.` $\exists$ `op'. M' ip op'`) is proved. We summarize a general method to prove the additional assumption of the design in Section 5.4.

5.3 Combining the Translator Correctness Theorems with the Importation Theorems

In this section, we introduce the basic idea about how to combine the translator correctness theorems with importation theorems based on a deep embedding semantics. This combination allows 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 result is obtained based on a low level data structure — an MDG representation: that is what the algorithms apply to. Therefore, the formalization of the MDG verification results in the importation theorems ought to be based on the semantics of the MDG representations. Moreover, 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 importation theorems which convert the low level MDG verification results into HOL to form the HOL theorems in terms of the semantics of *MDG-HDL*. In other words, we are not only able to import the MDG result into HOL based on a verified MDG system, but also the MDG verification results can be converted directly from the MDG input files to the theorems of HOL naturally.

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 7. In Figure 7, step 1 gives the 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* language (step 3). The MDG algorithm is then applied to the *MDG* in order to obtain two canonical *MDGs* and the MDG tool checks whether the two canonical *MDGs* are identical and returns true or false (step 4).

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 low level MDG result with 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 our work, we have verified the first translator. In order to demonstrate the combination of the translator correctness theorems and the importation theorems, the formalization of the MDG results will be in terms of the *core MDG-HDL*. In fact, the principle is the same. Similar conversion can be done for further verified translators. By combining the translator correctness theorem with the importation theorems, we obtain the new importation 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 allows 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*.

Therefore, the different MDG verification applications are formalized in a way that corresponds to the semantics of the low level program (*core MDG-HDL*) and converted into HOL to form the HOL theorem in terms of the semantics of *MDG-HDL*. We have obtained theorems for both combinational verification and sequential verification. They state that a verification result about a circuit based on a low level program is equivalent to a HOL theorem based on the semantics of the high level language (*MDG-HDL*), i.e., that the structural specification implements the behavioral specification.

We now describe how to obtain the new sequential verification importation theorem. That is, given an importation theorem about low level code, we use the translator correctnesss theorem to obtain one about high level code. We first

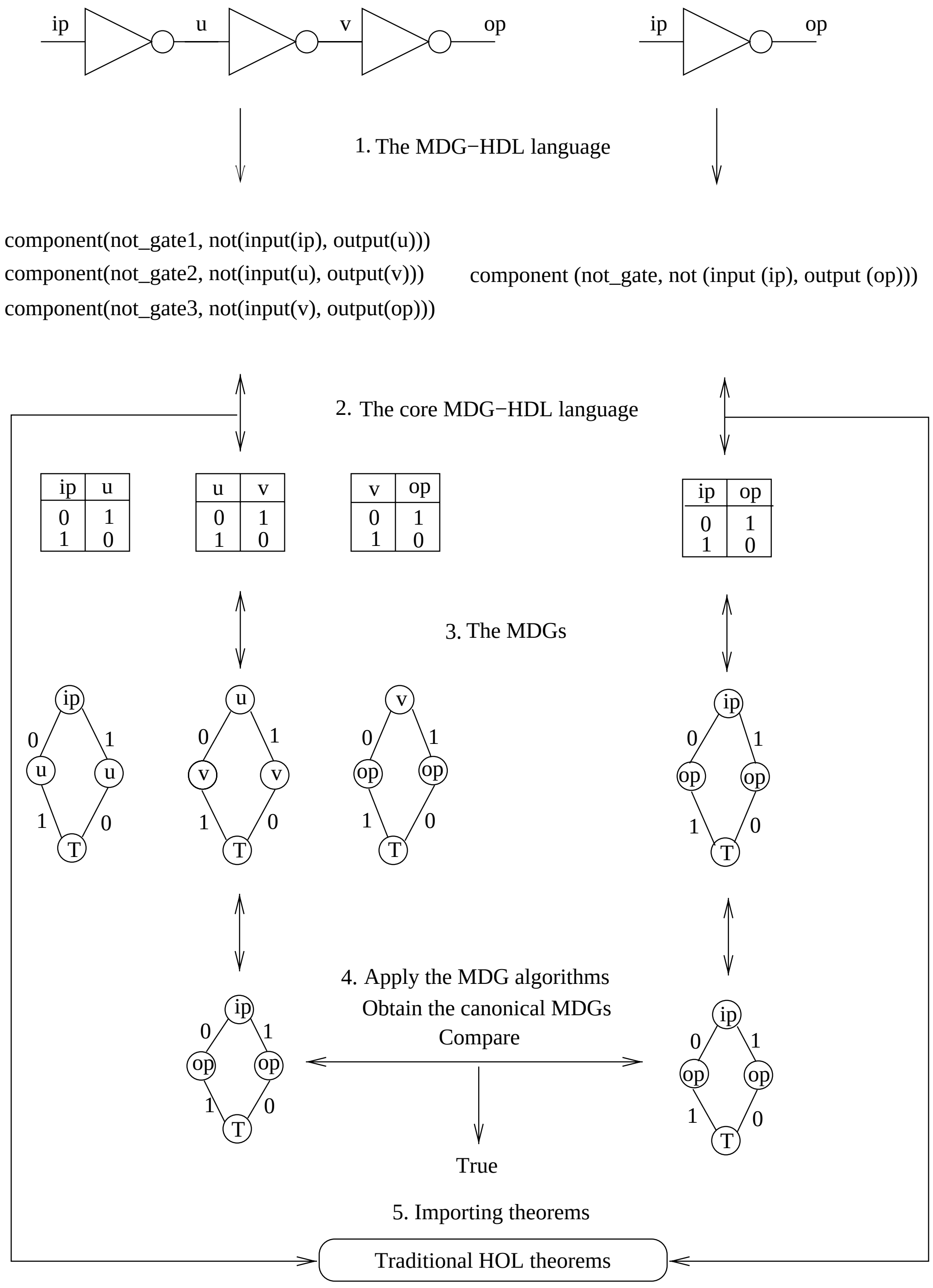
ip
u
v
op
ip
op
1. The MDG−HDL language
component(not_gate1, not(input(ip), output(u)))
component(not_gate2, not(input(u), output(v)))
component(not_gate3, not(input(v), output(op)))
component (not_gate, not (input (ip), output (op)))
2. The core MDG−HDL language
ip u 0 1 1 0
u v 0 1 1 0
v op 0 1 1 0
ip op 0 1 1 0
3. The MDGs
ip
u
v
op
T
0
1
4. Apply the MDG algorithms
Obtain the canonical MDGs
Compare
True
5. Importing theorems
Traditional HOL theorems

Fig. 7. The MDG Verification Process

instantiate the two machines in terms of the semantics of the *core MDG-HDL* language in the importation theorem (5). Therefore, we obtain the importation theorem based on the semantics of the *core MDG-HDL* language as shown below:

```
⊢thm ∀ IMP SPEC.
      (∀ ip op op' flag.
        PSEQ ip op op' flag
          (SemProgram_Core (TransProgMC SPEC))
          (SemProgram_Core (TransProgMC IMP))
                    ⊃ (∀ t. flag t = T)) ∧
     (∀ ip. ∃  op'. SemProgram_Core (TransProgMC SPEC) ip op') ⊃
     (∀ ip  op.(SemProgram_Core (TransProgMC IMP) ip op) ⊃
                (SemProgram_Core(TransProgMC SPEC) ip op))        (6)
```

Secondly, we need to prove the additional assumption.

```
⊢thm (∀ ip. ∃  op'. SemProgram_Core(TransProgMC SPEC) ip op')        (7)
```

From the translator correctness theorem (1), we obtain a theorem *Exist_Equ_Thm* (8). 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*. We discuss how this is done in Section 5.4.

```
⊢thm (∀ ip. ∃  op'. (SemProgram_Core (TransProgMC SPEC))) ip op') =
     (∀ ip. ∃  op'. SemProgram SPEC ip op')                          (8)
```

Thirdly, we prove a theorem, *Imp_Equ_Thm*, which states that the HOL theorem based on the semantics of the *core MDG-HDL* language is equivalent to that based on the semantics of *MDG-HDL*.

```
⊢thm (∀ ip op.
       (SemProgram_Core (TransProgMC IMP)) ip op  ⊃
        (SemProgram_Core (TransProgMC SPEC)) ip op) =
      (∀ ip op. (SemProgram IMP) ip op ⊃
                     (SemProgram SPEC) ip op)                 (9)
```

Finally, a new importation theorem *Import_Mdghdl_Thm* is obtained by rewriting theorem (6) with the theorems (8) and (9).

```
⊢thm ∀ IMP SPEC.
      (∀ ip op op' flag.
        PSEQ ip op op' flag
          (SemProgram_Core (TransProgMC SPEC))
```

$$\begin{array}{l}
\texttt{(SemProgram\_Core (TransProgMC IMP))} \\
\qquad\qquad \supset \texttt{(}\forall\texttt{ t. flag t = T))} \wedge \\
\texttt{(}\forall\texttt{ ip. }\exists\texttt{ op'. SemProgram SPEC ip op')} \supset \\
\texttt{(}\forall\texttt{ ip op. SemProgram IMP ip op} \supset \\
\qquad\qquad \texttt{SemProgram SPEC ip op)}
\end{array} \tag{10}$$

This result, a combination of the translator correctness theorem and importation theorems allows MDG verification results to be imported into HOL in terms of the semantics of *MDG-HDL*. An example of importing an MDG verification result into HOL will be given when we describe the case study in Section 6.

5.4 Proving the Existential Theorem

Above we proved the importation theorem for sequential verification. It has the form:

$$\begin{array}{l}
\vdash_{thm} \texttt{Formalized\_MDG\_result} \wedge \\
\qquad \forall\texttt{ ip. }\exists\texttt{ op. SPECIFICATION ip op} \supset \\
\quad \texttt{(}\forall\texttt{ ip op. (IMPLEMENTION ip op} \supset \texttt{ SPECIFICATION ip op))}
\end{array}$$

where *SPECIFICATION* represents the behavioral specification and *IMPLEMENTATION* represents the structural specification of a design. 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 *SPECIFICATION* can be satisfied for some outputs:

$$\vdash_{thm} \forall\texttt{ ip. }\exists\texttt{ op. SPECIFICATION ip op} \tag{11}$$

Similar existential theorems are also needed about implementations. When we convert MDG results into HOL to form HOL theorems, the theorems actually state that the implementation of the design implements its specification as shown in (12).

$$\vdash_{thm} \forall\texttt{ ip op. IMPLEMENTATION ip op} \supset \texttt{ SPECIFICATION ip op} \tag{12}$$

This representation might meet an inconsistent model that trivially satisfies any specification. This is sometimes called "The false implies anything problem" [5]. 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 [26], which has the form:

$$\vdash_{thm} \forall\texttt{ ip. }\exists\texttt{ op. IMPLEMENTATION ip op} \tag{13}$$

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. This has the same form (though for the implementation) as the importation theorem assumption has for the specification.

We have investigated a way of proving the additional assumption and the stronger consistency theorem based on the syntax and semantics of the MDG input language [32]. 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. As noted, the above two theorems have the same form. We call them *existential theorems.* The stronger consistency theorem (13) is an *existential theorem* for the structural specification, whereas the additional assumption (11) for the importation theorem is an *existential theorem* for the behavioral specification. If we use `C` to represent any specification or implementation of a circuit, and `ip` and `op` to represent the external inputs and outputs, *existential theorem* should have the form:

$$\vdash_{thm} \forall\ \texttt{ip}.\ \exists\ \texttt{op}.\ \texttt{C ip op} \qquad (14)$$

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

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

The *existential theorem* is existentially quantified. We can remove hidden lines in goals of this form using the HOL tactic `EXISTS_TAC`, which strips away the leading existentially quantified variable and substitutes a supplied *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 of 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. The HOL tactic *EXISTS_ELIM_TAC* [2] is used to eliminate existentially quantified variables in a goal. This tactic corresponds to a theorem *EXISTS_ELIM* given below.

$$\vdash_{thm} (\exists\ \texttt{x}.\ (\texttt{x = t}) \land (\texttt{A x})) = \texttt{A t} \qquad (15)$$

In other words, if the existentially quantified variable (`x`) is explicitly represented by its value as in (15) with $(\mathrm{x} = \mathrm{t})$ 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 which are not represented in the form $(\mathrm{x} = \mathrm{t})$, we need to find their *output representations.*

To import MDG results into HOL avoiding inconsistent models, we need to prove the *existential theorems* for each circuit. Although one is the *existential* *theorem* for the behavioral specification and one is the *existential theorem* for the structural specification, they all use the same representation language: the MDG input language (*MDG-HDL*) in our case.

We prove the existential theorems based on the syntax and semantics of *MDG-HDL* [32] [13]. However, a similar method can be used to solve other existentially quantified goals. We provide the *output representation* for each component (mainly logic gates and flip-flops). The *existential term* of a design, which reduces the goal `∃ x. t` to `t[u/x]`, is determined in terms of the corresponding *output representations*. This is very important for verifying the existential theorem, since as long as we find the *existential term* of the design, the corresponding theorem will be proved. We also provide HOL tactics for expanding the semantics of the circuit and proving the *existential theorem*. Further details can be found in [30] and [33].

6 Case Study: Integrated Hardware and Usability Verification

So far, we have discussed how to prove translator correctness theorems and importation theorems. Their combination allows the MDG verification results to be formalized and reasoned about in HOL in terms of the semantics of *MDG-HDL*. We now consider a simple case study to show that this method works in practice. We show how an actual MDG verification result can be imported into HOL to form a traditional HOL theorem. Moreover, we show that the importation theorems can be practically used in HOL. In particular, we apply our approach to a simple example, integrating MDG hardware verification and HOL usability verification for a vending machine.

The vending machine here is used to sell chocolate as shown in Figure 8. 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. In summary, the vending 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.

The vending machine is implemented in hardware as shown in Figure 9. We can use the predefined components in the MDG-HDL library to represent the corresponding circuit as described in [10]. In the circuit, two registers are needed to store the 4 internal states of the vending 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.

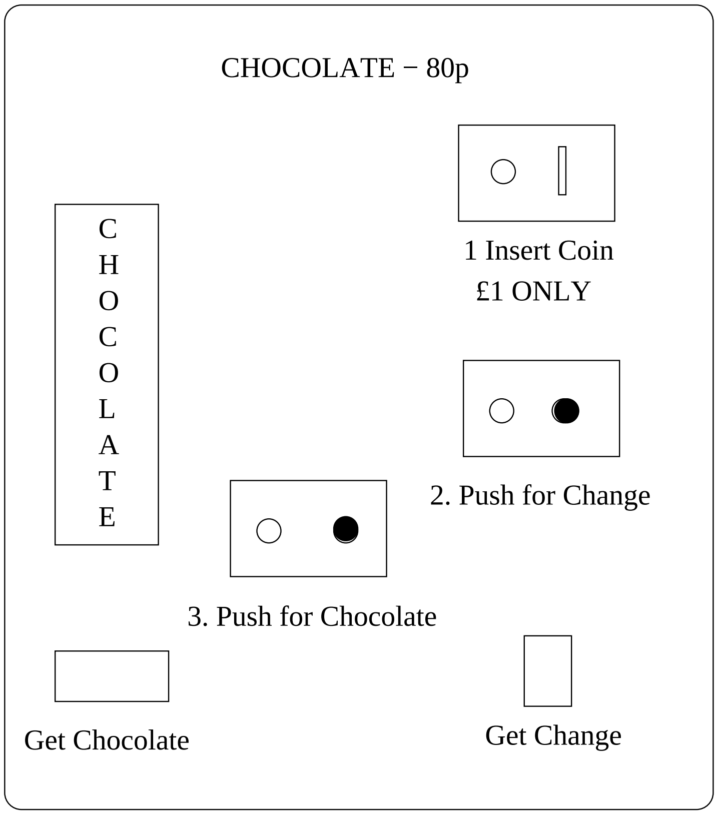

Fig. 8. The Vending Machine

This example was originally used to verify the absence of a common class of user errors known as post-completion errors (users missing completion tasks after their goal is achieved, e.g. forgetting to take change after getting chocolate.) within the framework of traditional hardware verification by Curzon and Blandford [11]. It was proved that the implementation of the vending machine meets its specification. A usability theorem about the absence of post-completion errors based on its *specification* was then proved. By combining the above two theorems, the usability theorem based on its *implementation* was proved. In the original work all the verification was done in HOL.

We closely followed their steps. However, we used the MDG system to verify the correctness of the vending machine and imported it into HOL using theorem (10). We then proved the *specification* based usability theorem in the HOL system. By combining those two theorems, first the correctness theorem of the vending machine which is verified in MDG (the importation theorem), second the *specification* based usability theorem which is proved in HOL, we obtained the *implementation* based usability theorem. Therefore, the importation theo-

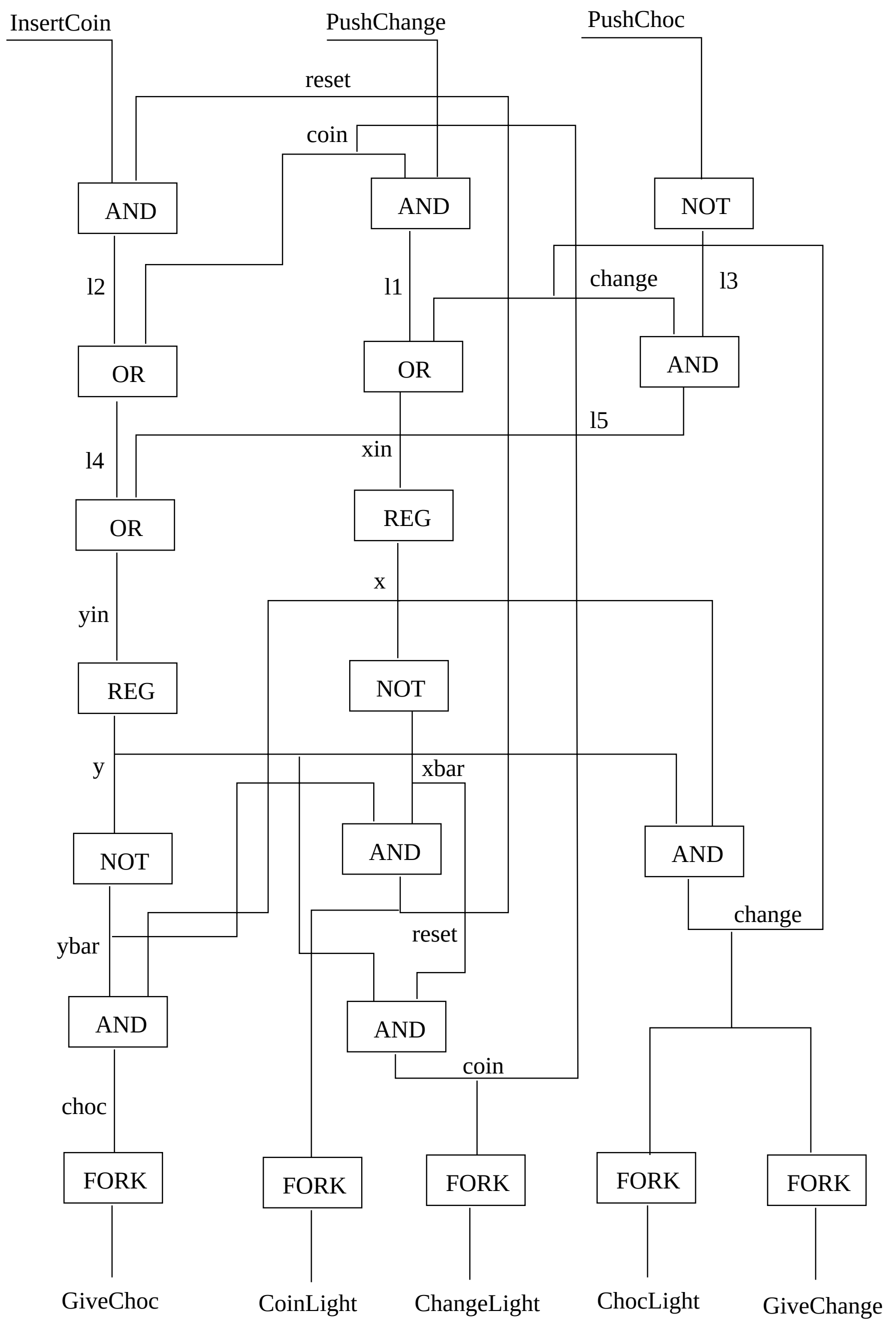

Fig. 9. The Circuit Implementation of the Vending Machine

rem (the correctness theorem) cannot only be imported into HOL but also can be used in HOL.

We first did a hardware verification of the vending machine in MDG. The theorem about the formalization of the MDG verification result can be tagged into HOL in terms of the semantics of *core MDG-HDL*.

$$\begin{aligned}
\vdash_{thm} & \ \forall\ \texttt{ip flag op op'}. \\
& \quad \texttt{PSEQ ip flag op op'} \\
& \qquad \texttt{(SemProgram\_Core (TransProgMC } \textit{Vend\_Imp\_Syn}\texttt{))} \\
& \qquad \texttt{(SemProgram\_Core (TransProgMC } \textit{Vend\_Spe\_Syn}\texttt{))} \\
& \qquad\qquad \supset (\forall\ \texttt{t. (flag t = T))}
\end{aligned} \tag{16}$$

where *Vend_Imp_Syn* and *Vend_Spe_Syn* stand for the syntax of the implementation and specification of the vending machine in terms of *MDG-HDL*. As stated in Section 5.3, the importation theorem for the vending machine can be obtained by instantiating the high level language importation theorem (10) with the syntax of its implementation and specification (*Vend_Spe_Syn* and *Vend_Imp_Syn*). We obtain theorem *Import_Vend_Thm*:

$$\begin{aligned}
\vdash_{thm} & \ (\forall\ \texttt{ip op op' flag}. \\
& \quad \texttt{PSEQ ip op op' flag} \\
& \qquad \texttt{(SemProgram\_Core (TransProgMC } \textit{Vend\_Imp\_Syn}\texttt{))} \\
& \qquad \texttt{(SemProgram\_Core (TransProgMC } \textit{Vend\_Spe\_Syn}\texttt{))} \\
& \qquad\qquad \supset (\forall\ \texttt{t. flag t = T})) \wedge \\
& \forall\ \texttt{ip.}\ \exists\ \texttt{op'. SemProgram } \textit{Vend\_Spe\_Syn}\ \texttt{ip op'} \supset \\
& (\forall\ \texttt{ip op. SemProgram } \textit{Vend\_Imp\_Syn}\ \texttt{ip op} \supset \\
& \qquad\qquad \texttt{SemProgram } \textit{Vend\_Spe\_Syn}\ \texttt{ip op})
\end{aligned} \tag{17}$$

Note that the first part of this theorem concerns the low level code produced by the translator, but that the existential assumption and conclusion are about high level code.

We then prove the *existential theorem* for the behavioral specification in terms of the semantics of *MDG-HDL*.

$$\vdash_{thm}\ \forall\ \texttt{ip.}\ \exists\ \texttt{op'. SemProgram } \textit{Vend\_Spe\_Syn}\ \texttt{ip op'} \tag{18}$$

Finally, the conversion theorem can be obtained by discharging the formalization theorem (16) and the existential theorem (18) from the importation theorem (17). This theorem states that the implementation of the vending machine implies its specification.

$$\begin{aligned}
\vdash_{thm} & \ \forall\ \texttt{ip op. SemProgram } \textit{Vend\_Imp\_Syn}\ \texttt{ip op} \supset \\
& \qquad \texttt{SemProgram } \textit{Vend\_Spe\_Syn}\ \texttt{ip op}
\end{aligned} \tag{19}$$

We next prove a *specification* based usability theorem about the vending machine in the HOL system. We will then use (19) to convert it into an *implementation* based usability theorem. The general user model for a vending machine is defined as `CHOC_MACHINE_USER ustate op ip`. This specifies a simple form of cognitively plausible user behavior. We prove that if a user behaves in this cognitively plausible way, they will not make post completion errors with the particular vending machine in question (though they may with other poorer designs). 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. The details of the user model are not important for our main argument here about integrating the results: the interested reader should refer to [11] [12].

```
⊢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 vending 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: they will not make post-completion errors.

```
⊢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 all the external inputs and outputs are Boolean, a user acts in the cognitively plausible way specified by the user model and the machine behaves according to its specification, then the usability property will hold.

```
⊢thm ∀ ustate op ip.
    Boolean ip op ∧
       CHOC_MACHINE_USER ustate op ip ∧
              CHOC_MACHINE_SPEC ip op ⊃
                      CHOC_MACHINE_USABLE ustate op ip        (20)
```

where predicate `Boolean` checks if all the external wires are Boolean values. This is required because the inputs of a `TABLE` could be either a concrete type variable or a Boolean variable. This predicate ensures the external wires have proper values.

The *implementation* based usability theorem can be proved in HOL by combining the correctness theorem (19) based on the MDG result with specification based usability theorem (20) proved in HOL. It (21) states that if the inputs and outputs are Boolean, a user acts rationally according to the user model and the machine behaves according to its *implementation*, then the usability property will hold.

```
⊢thm ∀ ustate op ip.
    Boolean ip op ∧
       CHOC_MACHINE_USER ustate op ip ∧
              CHOC_MACHINE_IMPL ip op ⊃
                      CHOC_MACHINE_USABLE ustate op ip        (21)
```

From this example, we have shown that a system can be verified in two parts. One part of the proof can be done in MDG and 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 importation theorems giving a greater degree of trust. We have made use of the importation theorem. In other words, the MDG verification result not only can be imported into HOL to form the HOL theorem, it can also be used as part of a compositional verification in HOL. We have also shown that two different applications (hardware verification and usability verification) suited to two different tools can be combined together.

7 Conclusions

We have described a methodology which provides a formal linkage between a symbolic state enumeration system and a theorem proving system based on verified symbolic state enumeration systems. The methodology involves the following three steps. The first step is to verify correctness of the symbolic state enumeration system in an interactive theorem proving system. Some symbolic state enumeration based systems such as MDG consist of a series of translators and a set of algorithms. We need to prove the translators and algorithms to ensure the correctness of the system. We have not verified the algorithms in this work, but concentrated on the translators. 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 those of its translated form. This work increases our trust in the results of the symbolic state enumeration system.

The second step of the methodology is to prove importation theorems in the proof 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 need to provide the theoretical justification for linking two systems.

The third step is to combine the translator correctness theorems with the importation 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 that the algorithms manipulate, and the result is strictly about, but imported in terms of the semantics of a high level language. 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.

We also summarize a general method to prove *existential theorems* of given designs, which strictly is needed for importing sequential verification results into a theorem proving system. This work makes the linking process easier and removes the burden from the user of the hybrid system.

We have implemented this methodology on a simplified version of the MDG system and the HOL system, and provided a formal linkage by using the above mentioned steps. We have verified aspects of correctness of a simplified version of the MDG system. We have provided a formal linkage between the MDG system and the HOL system based on importing theorems [31]. Most importantly, we have combined the translator correctness theorems with the importation theorems. The feasibility of this approach has been demonstrated in a case study: integrating hardware verification and usability verification for a vending machine. However, for importing the MDG verification result into HOL, we have to prove the *existential theorem* for the specification of the design. The behavioral specifications must be in the form of a finite state machine or table description.

In ongoing work we are verifying the translator from *core MDG-HDL* to *MDGs*. This has already been done, and integrated with the importation theorems, for a smaller subset of the language (only using Boolean sorts). We will also prove importation theorems for other MDG applications, verify more complex examples and use our method in a combined system. In the longer term we will verify the MDG algorithms and so integrate them with the existing work.

Acknowledgments
This work was 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, R. Kaivola, and C. J. H. Seger. Formal verification of iterative algorithms in microprocessors. *Design Automation Conference*, pages 201–206, June 2000. ACM Press.
2. 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.
3. R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions in Computers*, 35(8):677–691, August 1986.
4. R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computer Surveys*, 24(3):293–318, September 1992.
5. 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.
6. 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.
7. 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*, volume 1055 of *Lecture Notes in Computer Science*, pages 241–257. Springer-Verlag, 1996.
8. 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.
9. 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.
10. P. Curzon and A. Blandford. Reasoning about order errors in interaction. In *TPHOLs 2000 Supplemental Proceedings*, Technical Report CSE-00-009, pages 33–48. Oregon Graduate Institute, August 2000.
11. P. Curzon and A. Blandford. Using a verification system to reason about post-completion errors. In Philippe Palanque and Fabio Paternò, editors, *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*, pages 293–308, Limerick, Ireland, June 2000. Kluwer Academic.
12. P. Curzon and A. Blandford. Detecting multiple classes of user errors. In M.R. Little and L. Nigay, editors, *Engineering for Human-Computer Interaction, 8th IFIP International Conference, EHCI 2001*, volume 2254 of *Lecture Notes in Computer Science*, pages 57–71. Springer-Verlag, 2001.

13. 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.
14. L. A. Dennis, G. Collins, M. Norrish, R. Boulton, K. Slind, G. Robinson, M. Gordon, and T. Melham. The PROSPER toolkit. In *The Sixth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, pages 78–97. Springer Verlag, 2000.
15. M. J. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
16. M. J. C. Gordon. Reachability programming in HOL98 using BDDs. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1869 of *Lecture Notes in Computing Science*, pages 179–196. Springer-Verlag, 2000.
17. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-order Logic*. Cambridge University Press, 1993.
18. Orna Grumberg and Shmuel Katz. Veritech: Translating among specification and verification tools design principles. http://www.cs.technion.ac.il/labs/ssdl/research/veritech, March 1999.
19. J. Harrison and L. Théry. A skeptic's approach to combining HOL and Maple. *Journal of Automated Reasoning*, 21:279–294, 1998.
20. S. Hazelhurst and C. J. H. Seger. Symbolic trajectory evaluation. In T. Kropf, editor, *Formal Hardware Verification: Methods and Systems in Comparison*, volume 1287 of *Lecture Notes in Computer Science*, pages 3–79. Springer-Verlag, 1997.
21. A. Heck. *Introduction to MAPLE*. Springer, 1993.
22. J. Hurd. Integrating GANDALF and HOL. Technical Report 461, University of Cambridge, Computer Laboratory, April 1999.
23. J. Joyce and C. Seger. Linking BDD-based symbolic evaluation to interactive theorem-proving. In *the 30th Design Automation Conference*, pages 469–474, Texas, United States, 1993.
24. 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*, volume 2144 of *Lecture Notes in Computer Science*, pages 244–258. Springer-Verlag, 2001.
25. J Lind-Nielsen. BuDDY - A Binary Decision Diagram Package. http://www.itu.dk/research/buddy/.
26. T. F. Melham. *Higher Order Logic and Hardware Verification*. Cambridge Tracts in Theoretical Computer Science 31. Cambridge University Press, 1993.
27. L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.
28. S. Rajan, N. Shankar, and M. K. Srivas. An integration of model-checking with automated proof checking. In P. Wolper, editor, *Computer-Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97. Springer-Verlag, 1995.
29. T. Tammet. Gandalf version c-1.0c reference manual. http://www.cs.chalmers.sr/∼tammet/, October 1997.
30. H. Xiong. *Providing a Formal Linkage between MDG and HOL Based on a Verified MDG System*. School of Computing Science, Middlesex University, January 2002. Ph.D. thesis.

31. H. Xiong, P. Curzon, and S. Tahar. Importing MDG verification results into HOL. In *Theorem Proving in Higher Order Logics*, volume 1690 of *Lecture Notes in Computer Science*, pages 293–310. Springer-Verlag, 1999.
32. 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 Report CSE-00-009, pages 237–248, August 2000.
33. 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, September 2001.
34. Z. Zhou and N. Boulerice. *MDG Tools (V1.0) User Manual*. University of Montreal, Dept. D'IRO, 1996.