Providing a Formal Linkage between MDG and HOL

Haiyan Xiong[1], Paul Curzon[2], Sofiène Tahar[3], and Ann Blandford[4]

[1] Faculty of Science and Engineering,Manchester Metropolitan University, Manchester, UK.
h.xiong@mmu.ac.uk
[2] Queen Mary, University of London, London, UK
pc@dcs.qmul.ac.uk
[3] ECE Department, Concordia University, Montreal, Canada.
tahar@ece.concordia.ca
[4] UCL Interaction Centre, University College London, London, UK.
a.blandford@ucl.ac.uk

Abstract. We describe an approach for formally verifying the linkage between a symbolic state enumeration system and a theorem proving system. This involves the following three stages of proof. Firstly we prove theorems about the correctness of the translation part of the symbolic state system. It interfaces between low level decision diagrams and high level description languages. We ensure that the semantics of a program is preserved in those of its translated form. Secondly we prove linkage theorems: theorems that justify introducing a result from a state enumeration system into a proof system. Finally we combine the translator correctness and linkage theorems. The resulting new linkage theorems convert results to a high level language from the low level decision diagrams that the result was actually proved about in the state enumeration system.They justify importing low-level external verification results into a theorem prover. We use a linkage between the HOL system and a simplified version of the MDG system to illustrate the ideas and consider a small example that integrates two applications from MDG and HOL to illustrate the linkage theorems.

1 Introduction

Deductive theorem proving and algorithmic verification based 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. Algorithmic 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) [7].

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. Situations where control and data paths are mixed can also be problematic. An increase in complexity can result in the state space growing exponentially.

There has been a great deal of work concerned with combining theorem proving and symbolic state enumeration systems to gain the advantages of both. Shankar [40], for example, argues that significant breakthroughs can be achieved from the synergies between automated tools and interactive, deductive proof systems. A common approach to combining proof tools is to use a symbolic state enumeration system as an oracle that receives problems and returns answers to the theorem proving system. For example, the HOL system [25] 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.

Some symbolic state enumeration based systems such as MDG [15] consist of a series of translators and a set of algorithms. In essence there is a core verification system for a low-level language into which different high level languages can be translated for verification. 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.

This process raises several questions:

1. Does the core automated verification system produce correct results?
2. Do the translators correctly translate between the higher level languages used in the proof systems and the lower level languages used by the core algorithmic verification system?
3. Have the verification results from the automated verification system been correctly converted into a valid theorem in the theorem proving system?

We are not directly concerned with the answer to the first question here. Rather we are concerned with the linkage between the theorems in the theorem prover and the results proved by the core verification system. This entails both the latter two questions.

The main contribution of this paper is that we describe an approach for formally verifying the linkage between a symbolic state enumeration system and a theorem proving system. In particular the approach involves developing a series of theorems about the hybrid system.

– Theorems about the correctness of the translation part of the symbolic state system are proved. They interface between low level decision diagrams and high level description languages. It must be proved that the semantics of a program is preserved in those of its low level translated form.
– Generic *linkage theorems* are proved. They are theorems that justify introducing a result from a state enumeration system into a proof system. Their antecedent is a result of a form produced by the slave state enumeration system and conclusion of a form, such as an implication, of use in the master proof system. The generic linkage theorems can be instantiated with the semantics of a particular system's language to give concrete linkage theorems for that system.
– The translator correctness and concrete linkage theorems are combined to give new linkage theorems. They convert results about low level decision diagrams to results about the high level language programs. These theorems justify importing low level external verification results into a theorem prover.

To illustrate this approach, we look at how aspects of a linkage between the HOL system and a simplified version of the MDG system are verified based on the importing of MDG results to HOL theorems [46]. We combine translator correctness theorems with the linkage theorems in order to allow low level MDG verification results to be imported into HOL in terms of the semantics of the high level language *MDG-HDL*. Full details of this work can be found in [45].

We chose HOL and MDG specifically because this work is part of a collaboration with the Hardware Verification Group at Concordia University. They are developing an implementation of a practical hybrid system (MDG-HOL) [32] which combines the MDG system and the HOL system.

We verify a *specification* of a hybrid verification system rather than an implementation of a system in a similar way to e.g. Basin *et al* [4] and following a standard approach of compiler verification work from which this project in part derives. In this sense the verification is applicable to any implementation of an MDG–HOL linkage that meets the specification.

Basing our illustrative example on real verification systems rather than a toy combination ensures the work is grounded in reality. The intention is that the ideas have application to other systems which adopt a similar architecture too. Specifically the general approach is applicable to hybrid systems where the architecture involves the state enumeration system being called upon as an oracle to provide externally verified "theorems" to a theorem prover, and where the results are expressed in terms of high level languages translated down by the external systems into some low level language such as BDDs to which the reasoning algorithms apply.

By investigating a multiway decision diagram (MDG) based hybrid system we do not preclude the results applying to binary decision diagrams (BDDs) as MDGs extend BDDs. In fact an early subset considered was a boolean subset of a hybrid MDG–HOL system. We have not considered here the full MDG-HDL language as used in the Concordia MDG-HOL implementation, but a significant subset (as described subsequently) that is sufficient to illustrate the general approach to verifying a hybrid system linkage. We do not believe verifying a full specification would be problematic. We did not do so simply due to the limited time and resource constraints of our particular project.

Justifying whether the best hybrid system is one combining MDG and HOL as opposed to other pairs of systems (why HOL rather than ACL2? Why MDGs over BDDs? etc) is not the issue of this paper as we are concerned here specifically with the *verification* of such systems not specifically their use. However, we note that Zhou *et al* [50] showed the advantages of MDGs over BDDs using an island tunnel controller example. Whilst BDD checkers (SMV and VIS) were faster than MDG for fixed bit length examples, in MDG it was possible to verify a generic, parameterized n-bit version that was not possible with BDDs. As already discussed, there are potential benefits from combining automated and interactive verification tools in general. This is discussed for example by Tahar *et al* [41] who compared VIS, MDG and HOL. We review the literature on combining systems in detail in the next section.

We consider a small example that integrates two applications to illustrate the linkage theorems developed here: hardware verification (in MDG) and usability verification (in HOL). The linkage theorems are applied directly to the MDG results giving a HOL hardware correctness theorem that can be combined with the HOL usability correctness theorem. In this way a single HOL theorem is proved that integrates the two results. This example is given for illustration of the linkage theorems only. We do not envisage that a practical verification would be done in this way. The results from our approach would rather be used as a justification for increased trust in an implementation of a combined system such as the MDG-HOL system developed at Concordia University.

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 contribution of the paper describing the linkage theorem approach, illustrated with a linkage between HOL and MDG, is described in Section 5. Section 6 illustrates the linkage theorems developed by looking at a very simple application that involves combining two verification applications from MDG and HOL. Finally, our conclusion and suggestions for further work are presented in Sections 7 and 8. The HOL files related to the work in this paper giving full formal specifications and proof developments including semantics of the languages, definition of the translator, development of the linkage theorems and the example proof development can be accessed from: http://www.dcs.qmul.ac.uk/∼pc/research/mdg/program.zip

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 most connected to the kind of linkage we are verifying. We concentrate on higher-order logic proof systems since that is the main focus of this paper. A more general review of combined systems is given by Uribe [44].

Joyce and Seger [31] presented a hybrid verification system, HOL-Voss, which links HOL to a symbolic trajectory system, VOSS [28]. Predicates were defined in the HOL system, which created a mathematical link between the specification language of the Voss system and that of the HOL system. Aagaard *et al.* built on this work in developing the Forte verification system [1]. Forte combines model checking in Voss and the theorem proving system, ThmTac. ThmTac is written in `fl` (a strongly-typed functional language in the ML family [37]) 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. More recently, with industrial take-up at Intel, Forte [34] has become one of the most mature formal verification environments based on tool integration. It has been used in large-scale industrial verification projects at Intel. Its power comes from the very tight integration of the two provers based on reflection and using a single functional language as both the theorem prover's meta-language and its object language.

Rajan *et al.* [38] proposed an approach for the integration of model checking with PVS [16]: 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. Shankar [39] gives a more recent state of the art with respect to integrating automated tools of various kinds with PVS [40]. Shankar [39] also argues that model checking and deductive proof systems are best used to generate properties of specifications rather than to prove correctness as we do here. SAL [5], a Symbolic Analysis Laboratory, has been developed to further this approach providing a framework for combining different analysis tools around a single specification language.

Computer algebra tools are another class of tool that can provide useful automated results for a theorem prover. Harrison and Théry [27], for example, combined the theorem prover system (HOL) and the Maple [29] 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.

Adams *et al* [2], also implemented an interface between Maple and a theorem prover, here PVS. Their work shows an alternative approach – to have the theorem prover as the black box tool. PVS provided automated support to the computer algebra system, allowing it to prove, for example, side conditions.

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 [22] 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 [32] used the PROSPER/Harness Plug-in Interface to link the HOL system and the MDG system.

The VeriTech project [26] 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.

It is clear from the above discussion that there are many ways to link systems, from the direction of the control, how tight the linkage is and whether the linkage is a general infrastructure. We are not specifically concerned here with the development of combined systems, but with their verification, and more specifically the verification of the linkage. We are therefore specifically concerned with ways of increasing trust in linked systems.

One approach to trusting proof systems is that adopted by LCF [23]: having a small, trusted core of primitive inference rules, with all proof implemented ultimately as calls to these primitives. If the core is correct the theorems proved by the combined system can be trusted. One approach to the linkage of systems is therefore just to implement a linked system as calls to the primitive inferences. The type system then guarantees soundness of the combined system.

In work such as that described 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). Shankar [40] argues that in the near future it will be both 'prudent and feasible' to verify the deductive algorithms to be combined thus arguing that fully expansive approaches are not needed in the long term. Our interest is in the use of verification to increase the trust in linkages (where we focus here on the verification of the linkage rather than the algorithm). However, increased trust can be obtained in other ways to verification as we now briefly consider.

Gordon [24] investigated a way that such trust can be increased. He integrated the BDD based verification system BuDDY [33] 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.

Amjad [3] continued in this line, developing a model checker within HOL. The model checking algorithm is implemented using HOL proof. Each model checking step is based on a theorem proved in HOL. However, where BDD computation is needed, calls are made to BDD primitive inference rules embedded in the HOL logic. The logic is thus extended with these extra primitives. In essence in this approach the interface between the trusted theorem prover and external tool is pushed downwards so that as much as possible is done within the tool. Empirical evidence suggests that the efficiency loss in this approach is within reasonable bounds. However the comparison was with a model checker implemented in ML rather than a standard model checker and only on a small example. This approach also still leaves results reliant on the soundness of the underlying BDD tools, though they are much simpler than a full external tool. A high assurance of soundness is obtained at the expense of some efficiency. Whereas this approach involves writing a new tool embedded in the theorem prover to give the trusted linkage, our work focusses more on how one can raise the level of assurance of existing tools that can then be used as black box provers, thus avoiding any loss of efficiency.

Mhamdi and Tahar [36] follow a similar approach to the BuDDY work developing invariant checkers and model checkers in HOL but deeply embedding MDGs rather than BDDs. This work builds on the MDG-HDL [32] project, but uses a tightly integrated system with the MDG primitives written in HOL rather than two tools communicating as in the MDG-HOL system that we verify aspects of here. The potential advantages of combining HOL and MDG are also illustrated by the island tunnel controller study [36]. The shallow linked tools approach provides a slightly worse CPU time and better memory usage than the MDG tool alone. However, with the deeply embedded tools method, the CPU time is much slower but memory usage is significantly better.

Hurd [30] used a different method to combine the strengths of two systems without loss of trust. He considered two theorem-prover systems—Gandalf [42] 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. Harrison and Théry's architecture [27] combining HOL and Maple noted above similarly used Maple to find witnesses that could then be used in fully expansive HOL proofs so avoiding compromising the soundness of their results.

Such approaches do increase the trust, but do so at the expense of efficiency at least to some extent. An alternative is to take a standard linkage and use a deductive theorem prover to verify the algorithmic model checker as noted by Shankar [40]. This of course can be done independently of whether the systems are to be combined.

Early work in this area is Chou and Peled's work [14]. They verified correctness properties about the preservation of safety and liveness properties of the partial-order reduction technique used by SPIN for model checking. This proof was done using the HOL verification system.

Théry [43] has performed a machine-checked verification using Coq of Buchberger's Gröbner bases algorithm used to solve algorithms in commutative algebra and contained in all major computer algebra software systems. Buchberger has gone further [9], outlining a way the algorithm could be automatically synthesised.

Basin *et al* [4] were also concerned with the verification of model checkers though used for a specific application: to verify bytecode. As with our work, rather than verifying model checker implementations they are in essence verifying specifications of model checkers. They verify the 'approach'. The essence of the theorem proved is that if their declarative model checking specification is followed then the bytecode it was applied to has a well-typing; i.e. it is free from runtime errors. They argue that by verifying the specification, the work applies to a range of different model checkers implemented from it, hence the claim that they have verified multiple model checkers – though no claim is made that actual implementations have been verified. The theorem proved is a form of linkage theorem in our terms in that it links the result of a model checker to a result about a specific domain: the type checking of bytecode. However, the focus of this work is not as ours strictly about the verification of general linkage results from an external tool. The application area also differs in that we specifically consider modular, compositional hardware verification systems in this paper.

The above work suggests that the machine-checked verification of external algorithms linked to interactive systems is likely to be feasible in the longer term. Here, as with the work above as noted, we are considering a step towards the verification of full implementations in that we verify a specification of a hybrid verification system.

In our approach, 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 (using the MDG system as exemplar) and provide a theoretical underpinning to the formal linkage of such a system and a theorem proving system (MDG and HOL are used here). 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 linkage theorems, the MDG verification results can be imported into HOL in terms of the MDG input language (*MDG-HDL*). We illustrate the correctness theorems developed using a concrete example 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.

There is no single right approach to increasing trust in hybrid verification systems. Ultimately in the long term, if it proves practical, having fully formally verified implementations of linked systems would seem to be most desirable, as it avoids loss of efficiency in using the tool. This is an important issue if formal verification tools are to be used industrially as standard. However, this is still beyond the state of the art, and may not be desirable for all verification systems. In some cases, for example where the aim of using formal verification tools is to find bugs, not give high assurance that there are none, the fact that the tools used are not completely sound may not be a major issue. Furthermore, in the short term fully verified hybrid systems do not yet exist so there are clear advantages of gaining trust in other more immediately tractable ways such as the LCF/BuDDY approach or that of using the external tool to provide witnesses from which to build a proof.

Ultimately there is a trade-off between the time expended in verifying verification tools and the time saved by then having a more efficient tool that is trusted. If the core of a verification system stabilizes then it is worth the effort invested in verifying it. Whilst verification systems are still research vehicles undergoing continued development, with different combinations of hybrid systems being experimented with and new more powerful external verifiers being added, the advantages of verifying any one combination are reduced. However, the leverage in verifying verification systems in general is great as the benefit of the verification of a tool spreads to all uses of it. Research into such approaches is thus very important. Whilst the verification of a tool specification as in our example here does not give the same amount of trust as verifying an implementation, it is a useful step on the way that does give an additional level of trust. A similar issue applies to the fact that we do not consider here the verification of the decision diagram algorithms. Without that step the trust we can have in the soundness of the system is reduced greatly. However, that does not preclude the benefit gained from verifying the other aspects.

3 The MDG System

The MDG system [15] 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) [8] 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 $\vee_{1 \leq i \leq n} \; ((A = B_i) \wedge P_i)$.

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 circuits that can be proved.

The MDG package [49] is 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*) [49], 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 [25] 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 [23] 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 [37], 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.

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

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 the axiom definition facility are not used.

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.

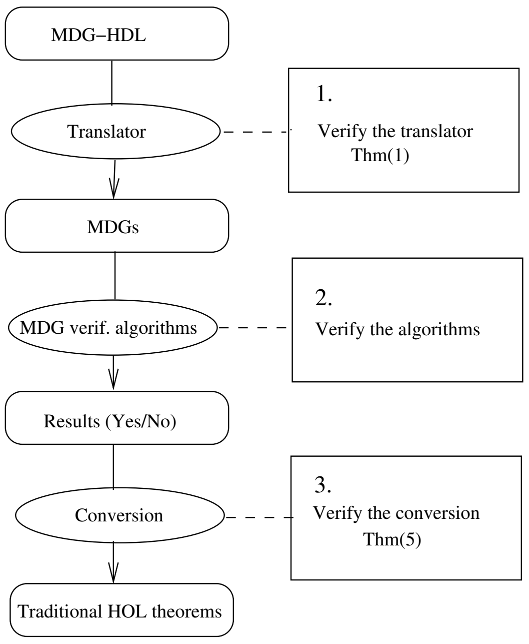

Fig. 1. Overview of the Formal Linkage Project

5 Formally Linking a Verified MDG and HOL System

Our work explores a way of increasing the degree of trust of linked systems. To illustrate the approach we consider the verification of a linkage between the HOL system and the MDG system in terms of the MDG input language as shown in Figure 1. This work is 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) [47] and (2) verification of the algorithms (step 2 of Figure 1).
- We then must prove theorems which we refer to as "linkage theorems" (step 3 of Figure 1), which formally convert the verification results of the MDG applications into traditional HOL hardware verification theorems [46].
- By combining the correctness theorems (theorems obtained from step 1 and step 2 of Figure 1) about the MDG system with the linkage theorems (obtained from step 3 of Figure 1), 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, importing the MDG results into HOL to form the HOL theorems (step 3 in Figure 1) [46] and how the two are combined. Chou and Peled's work [14] verifying a partial-order reduction technique for model checking is an example of research concerned with Step 2. Verifying the algorithms is beyond the scope of this paper: we are primarily concerned with the linkage and how correctness theorems about an external system, as exemplified by the translation theorems, can be combined with linkage theorems.

We use a "deep embedding" [6] of the MDG-HDL and other intermediate languages. In particular this means that we introduce the abstract syntax of expressions as a new higher-order logic type and then define the semantics of expressions for this syntax within HOL. This contrasts with a "shallow embedding" where the abstract syntax is not formally represented in the logic, only in the meta-language. In general, a deep embedding allows one to reason about the language itself rather than just the semantics of programs in the language. In our work this is important as we wish to reason about translators that act on the syntax. In particular it means we can prove separate translator correctness theorems and combine them with each other and with the linkage theorems.

As an aside to this main thrust of the work, we also summarize a general method for proving stronger consistency theorems, which occur as existential assumptions to the linkage theorems. These tools remove the burden from the user of a combined system [47] in proving such theorems manually. They are needed specifically to justify importing sequential verification results on sequential designs into the theorem proving system.

In the remainder of this section, we discuss the individual steps that we have undertaken: verifying the translator correctness theorems, proving the general linkage theorems, instantiating the linkage theorems with the semantics of a particular system's languages, combining the translator correctness theorems with the linkage theorems and discharging the existential assumptions.

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 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 library, 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 If-Then-Else 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`.

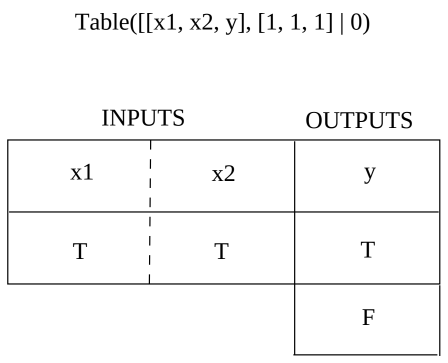

Fig. 2. The AND Table

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

Fig. 3. Overview of the MDG Translation Phases

We specify the translation of the MDG-HDL program in stages (Figure 3). The library components are each first compiled into a table representation. This is possible because the Table construct provides a general specification mechanism. We refer to resulting programs that use only the TABLE construct in this way as being *core MDG-HDL* programs. The resulting *core MDG-HDL* program can then be compiled into an internal *MDG*.

Adopting this approach makes the translation phase more amenable to verification. Note we are not verifying the actual MDG implementation. Although an MDG system could be implemented in this way, the existing implementation is not - whilst most components are translated into tables, some components, such as registers, are implemented directly in terms of an *MDG* for efficiency. However, 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 [13] with respect to compiler correctness.

As part of the translator verification we define a deep embedding semantics of a subset of the *MDG-HDL* language. This subset includes all of MDG-HDL except for three predefined components: the Multiplexer, the Driver and the Transform construct used to apply functions. These components are omitted from our subset, even though they are implemented in the actual MDG system, as they have non-boolean inputs or outputs. We consider this subset since our aim is to explore the feasibility of our approach. The subset considered does allow

$$\begin{tikzcd}
\begin{array}{c}\text{MDG–HDL}\\ \text{Syntax}\\ \textit{p}\end{array} \arrow[r, "\textit{TransProgMC}"'] \arrow[d, "\begin{array}{c}\text{MDG–HDL}\\ \text{Semantics}\end{array}"'] & \begin{array}{c}\text{core MDG–HDL}\\ \text{Syntax}\\ \textit{TransProgMC p}\end{array} \arrow[d, "\begin{array}{c}\text{core MDG–HDL}\\ \text{Semantics}\end{array}"'] \\
\begin{array}{c}\text{MDG–HDL}\\ \textit{SemProgram (p)}\end{array} \arrow[r, leftrightarrow, "\text{Thm(1)}"'] & \begin{array}{c}\text{core MDG–HDL}\\ \textit{SemProgram\_Core (TransProgMC p)}\end{array}
\end{tikzcd}$$

Fig. 4. Compilation Correctness

a program to contain concrete sorts. In other words, the inputs and outputs of a table could be of boolean sort or of concrete sort. 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. In the rest of this paper, we will refer to this simplified version of the MDG system as "the MDG system".

In the rest of this subsection, we describe the verification of the first translation step of the MDG system (phase 1 in Figure 3) based on the syntax and semantics of a subset of the MDG input language (*MDG-HDL*) and the *core MDG-HDL* language, using the HOL theorem prover. The syntax and semantics of these languages are defined. A translation function `TransProgMC` is defined that translates each *MDG-HDL* program to give the corresponding *core MDG-HDL* program (see Figure 4). The correctness theorem for this translation has been proved in HOL. It states that the semantics of the low level *core MDG-HDL* program is equal to the semantics of the high level *MDG-HDL* (the MDG input language).

The Syntax of the MDG-HDL Language The full abstract syntax of the subset of the MDG-HDL language used is given in the Appendix. The MDG-HDL commands consist of predefined MDG-HDL components, an operation to set the initial value of a variable, a next state variable command, a composition operation and a localisation operation. The syntax of this language introduces a specially-defined recursive data type of *component terms* to provide an explicit representation in logic of the MDG-HDL commands. We define a recursive type `Mdg_hdl` with 34 constructors for this purpose. The first 28 constructors are gates, flip-flops and registers. For example, the circuit term '`NOT ip op`' represents a *NOT* gate with one input labelled *ip* and one output labelled *op*.

A concrete sort is a set of distinct constants of that sort. We use a `string` to represent them. As the inputs and outputs of many basic components in the MDG-HDL library are of boolean value, we define a new type `Mdg_Basic` in HOL that represents both boolean and concrete sorts.

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

The constructor `INIT` represents the initial value of a state variable. `SNXT v nv` states that `nv` is the next state variable of the state variable `v`. If `c1` and `c2` are two values of type `Mdg_hdl`, then the term `JOIN c1 c2`, represents the composition of the two terms represented by `c1` and `c2`. The `INTERNAL` constructor represents the localisation operation. If `c` is a term representing a circuit and `w` is a string (internal wire), then the circuit '`INTERNAL w c`' represents the circuit obtained by hiding the wire labelled `w` in the circuit represented by `c`.

The constructor `TABLESYN` represents the syntax of the Table component which has five arguments. The first argument is a list of input variables. The second argument is the single output variable. This output could be either a current state variable or a next state variable. We define a new HOL type `out_type` to represent these options:

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

The third argument of the constructor `TABLESYN` is a list of table rows. Each row is a list itself giving one allocation of values to the inputs, and each input is an `Mdg_Basic` term. In other words, an input could be a (`BOOL bool`) term or a (`CONCRETE string`) term. The entries in the list can be either actual values or a special don't care marker. This is realised by defining a new type (as given in [21]).

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

TableVal_to_Val (TABLE_VAL (v:α)) = v
```

The fourth argument of `TABLESYN` has the type of `((Mdg_Basic) list)`. Each element of this argument gives the output for the corresponding allocation of values to the inputs. The final argument is the default value, taken by the output if the input values do not match any row. The default value could be an arbitrary `Mdg_Basic` value, a current state variable or a next state variable. Again we define a new HOL type *default_type* in terms of the type *out_type*.

```
default_type = DENORMAL of Mdg_Basic |
               DEOUT of out_type|
```

The abstract syntax of an MDG-HDL program as a whole is given by the constructor `PROG`. It consists of an external output wire list `Exoutput`, an external input wire list `Exinput`, an internal wire list `Invariable` and a *component term*, `Mdg_Hdl`.

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

The Syntax of the Core MDG-HDL Language Core MDG-HDL is a minimal subset of MDG-HDL using TABLES in place of the library components. The abstract syntax of the core MDG-HDL language is defined in terms of four arguments – an external output wire list, an external input wire list, an internal wire list and a *core component term*. A *core component term* consists of four constructors. i.e. `INITC`, `SNXTC`, `TABLESYNC` and `JOINC`.

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

The abstract syntax of a whole core MDG-HDL program is:

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

Translating MDG-HDL into the Core MDG-HDL Language For the translator we first define a set of functions that given a component return its core MDG-HDL code. For example, a `NOT` gate is translated into

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

We then define a function `TransGT` inductively over the syntactic structure of MDG-HDL component terms translating each into the equivalent core MDG-HDL form.

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

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

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

The Semantics of the MDG-HDL Program In this section we outline the semantics of MDG-HDL programs used in the verification. We first define semantic functions for each component in the MDG-HDL component library. We then define the semantics of MDG-HDL component terms (`SemMdghdl`). Finally, we define the semantics of MDG-HDL programs as a whole (`SemProgram`).

The primitive components of MDG-HDL component terms are logic gates, flip-flops, table, initial value etc. The semantics of logic gates and flip-flops have different sorts. We must first define predicates that ensure that each variable does not have mismatched sorts. For example, it is meaningless for a `NOT` gate to have non-boolean input. We need to check if the input or output for the different components and different applications is either a `BOOL bool` term, a `CONCRETE string` term or an `UNBOUND` term as appropriate.

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

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

where predicate `IS_BOOL` is used to check if a `Mdg_Basic` term's value is `BOOL T` or `BOOL F`, and `MDG_TO_BOOL` converts the `Mdg_Basic` terms `BOOL T` and `BOOL F` to boolean values `T` and `F`.

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

The semantics of the other logic gates, flip-flops and table are defined similarly.

The semantics of MDG-HDL *component terms* is defined as a higher-order logic function `SemMdghdl`:

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

Finally, the semantics of the program is defined in terms of the environment. It maps a syntactic object to a history function of type (`num`→`Mdg_Basic`). Function `Dsem_Ext` adds an extra entry to this environment for each external wire (input and output). A list `ip` is used to represent all the values of the external inputs and a list `op` is used to represent all the values of the external outputs. The semantics of the program can then be represented explicitly in terms of the external inputs `ip` and outputs `op`. The function `Dsem_Int` uses existential quantification to hide local variables from the environment. The entries for internal variables are added to the environment. The function `Check_External_Sort` makes sure that the external wires do not get sort mismatched. The semantics of the MDG-HDL program is defined in terms of these functions.

```
⊢def SemProgram (PROG exoutput exinput inv c) ip op =
     let env1 = (Dsem_Ext (SemExinput exinput) EmptyEnv ip)
```

```
    in
      let env2 = Dsem_Ext (SemExoutput exoutput) env1 op
      in
         ((Check_External_Sort c env2 (SemInvariable inv)) ⊃
            Dsem_Int (SemInvariable inv) c env2)
```

The Semantics of the Core MDG-HDL language The semantics of core component terms (`SemMdghdl_Core`) is similarly defined in terms of the semantic function for each component.

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

The semantics of core programs is also defined from auxilliary functions in a similar way to MDG-HDL itself. `Dsem_Int_Core` gives the semantics of the circuit in terms of the semantics of core component terms (`SemMdghdl_Core`) and uses existential quantification to hide the local variables from the environment of the circuit. The function `Check_External_Sort_Core` finds the proper sort for the external wires of the circuit. The semantics of the core MDG-HDL language is then defined as follows:

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

Translator Correctness Theorem We are now in a position to verify the correctness of the translator as we outlined at the start of this section. We have formally defined in higher-order logic the syntax and semantics of the two languages as well as the translation between them. We now have to obtain a theorem that quantifies over its syntactic structure stating that the semantics of the MDG-HDL program is equivalent to the semantics of the Table program it is translated to. We have proved a HOL theorem that states this formally:

```
⊢thm ∀ exv exi inv c.
       SemProgram (PROG exv exi inv c) ip op =
            SemProgram_Core (TransProgMC (PROG exv exi inv c)) ip op (1)
```

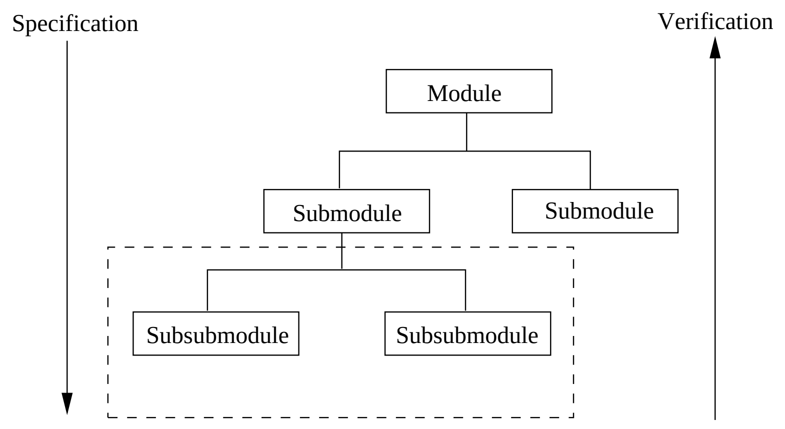

Fig. 5. Hierarchical Verification

This correctness theorem is straightforward to prove by structural induction on the syntax domain of the MDG-HDL program. For each component a separate proof shows that the semantics of the component and the semantics of the table that results from expanding the definition of the translator are equivalent. It is long-winded given the number of cases, but essentially straightforward.

5.2 The Linkage 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 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:

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

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. 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 MDG verification results into HOL, we have 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

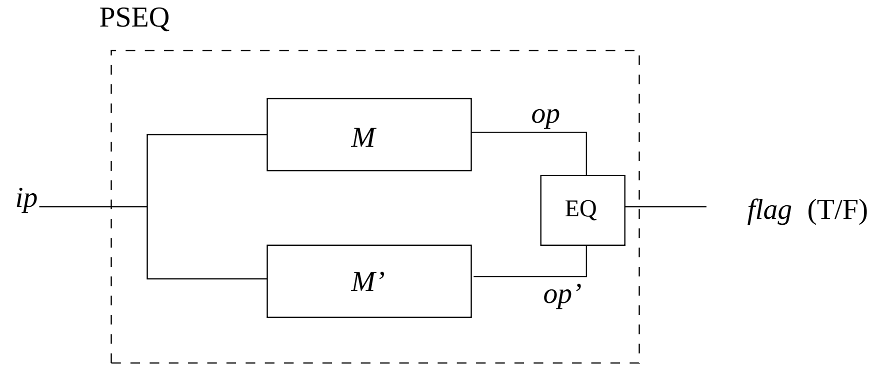

Fig. 6. The Product Machine used in MDG Sequential Equivalence Checking

form and so on. The most natural and obvious way to formalize the MDG results is in terms of what is in effect virtual hardware constructed so as to give a true output if given equivalent circuit descriptions. This does not give theorems of the form (an implication of those specifications) that we can easily integrate into a hierarchical proof. Therefore, we have to convert the MDG results into a form that can be used. We have proved a series of general linkage theorems that express the conversion produced by several different MDG verification applications These linkage theorems have the structure:

$$\begin{aligned} &\mathit{formalized\_MDG\_result} \supset \\ &\qquad\qquad \mathit{implementation} \supset \mathit{specification} \end{aligned} \tag{3}$$

The linkage theorems are general in the sense that they do not explicitly refer to the *MDG-HDL* semantics or multiway decision graphs. Rather they are given in terms of general relations on inputs and outputs. As they are independent of any specific language, the theorems proved could therefore be applied to other verification systems with similar architectures based on reachability analysis or equivalence checking. For example, suppose the behavioral equivalence of two state machines (Figure 6) is verified by some system (as it is in MDG) 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 proved (e.g. by MDG) about `PSEQ` is that the `flag` output is always true. This can be formalized as

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

We have proved the corresponding linkage theorem which converts such results to an implicative form:

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

Any verification system that formulates sequential verification as a product machine as in Figure 6 could use theorem 5 directly as the definition of `PSEQ` just specifies the structure of a general product machine. In particular, as MDG does it this way, this theorem can be used as an MDG linkage theorem.

Note this theorem 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 for proving this additional assumption of the design in Section 5.4.

In some cases, as here, equivalences rather than implications could be proved. For the above this would require an additional existential clause. As not all MDG applications verify equivalence all the linkage theorems we have developed are of an implicative form for consistency. It would be trivial to extend them to equivalences where the situation was symmetrical as with `PSEQ`.

5.3 Combining the Translator Correctness Theorems with the Linkage Theorems

In this section, we introduce the basic idea about how to combine the translator correctness theorems with linkage 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 linkage theorems ought to be based on the semantics of these low level MDG representations. We show here how the theorem about the translator's correctness can be used to overcome this problem. By combining the translator correctness theorems with the linkage theorems, we obtain new linkage theorems which convert low level MDG verification results into HOL theorems in terms of the semantics of the high level *MDG-HDL*.

The whole MDG verification and linking process is as illustrated in Figure 7, checking that three *NOT* gates are equivalent to a single *NOT* gate. Step 1 of

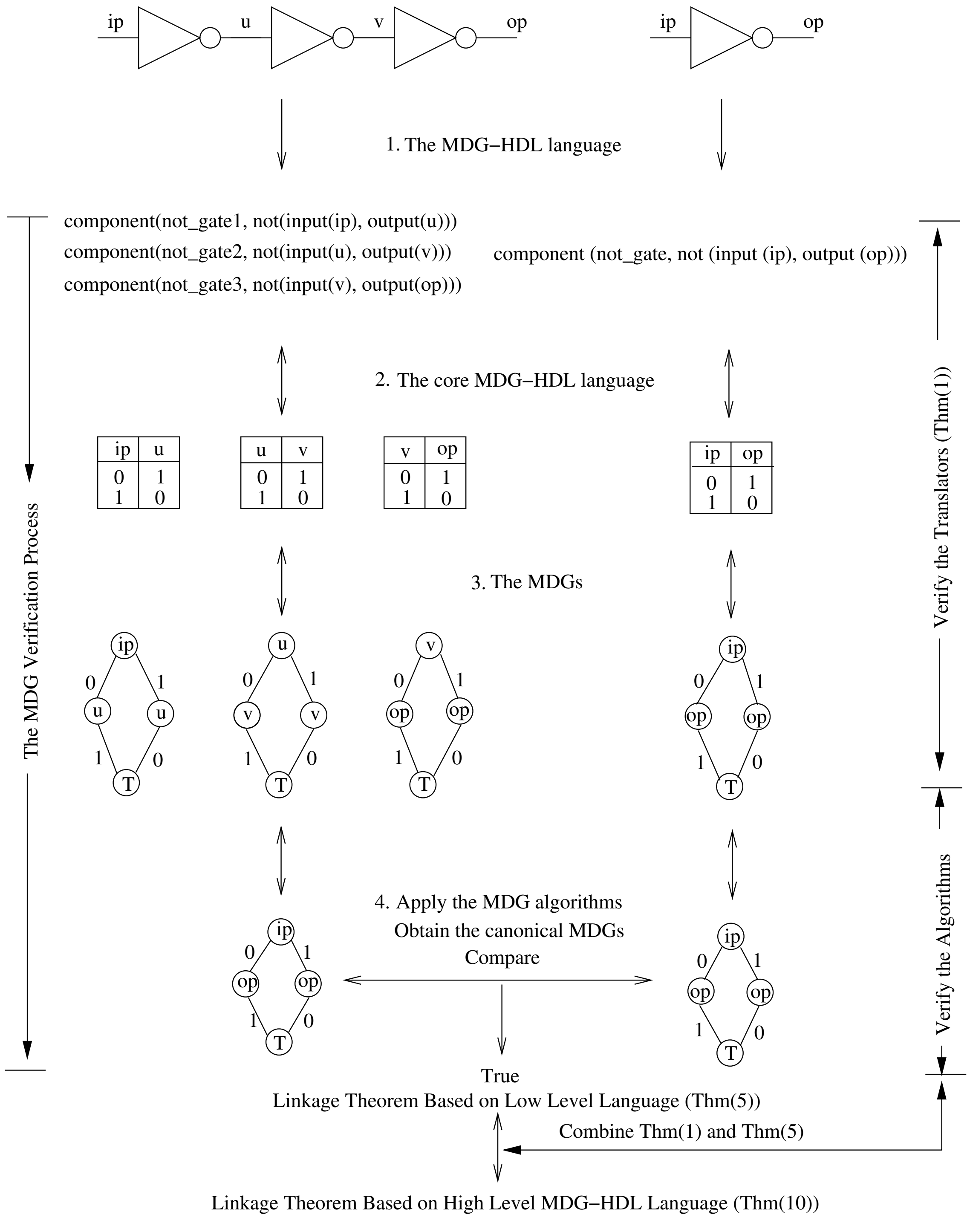
ip
u
v
op
ip
op
1. The MDG−HDL language
component(not_gate1, not(input(ip), output(u)))
component(not_gate2, not(input(u), output(v)))
component(not_gate3, not(input(v), output(op)))
component (not_gate, not (input (ip), output (op)))
2. The core MDG−HDL language
3. The MDGs
4. Apply the MDG algorithms
Obtain the canonical MDGs
Compare
True
Linkage Theorem Based on Low Level Language (Thm(5))
Combine Thm(1) and Thm(5)
Linkage Theorem Based on High Level MDG−HDL Language (Thm(10))
The MDG Verification Process
Verify the Translators (Thm(1))
Verify the Algorithms

Fig. 7. The MDG Verification Process illustrated by proving that three *NOT* gates are equivalent to a single *NOT* gate

the figure gives the main part of the two circuit description files (in 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 programs are then translated into the low level MDG language (step 3). The MDG algorithm is applied to the low-level MDGs in order to obtain two canonical MDGs. 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. From this result a HOL theorem is created.

The MDG verification results are about the low level *MDGs* that the algorithms manipulate 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 converted into HOL ones based on the semantics of the high level language (*MDG-HDL*). The combination also allows the additional assumptions that need to be proved for sequential verification to be proved in terms of the semantics of the high level *MDG-HDL*. In this paper we consider the verification of the first translator only so here the low-level formalization of the MDG results is in terms of *core MDG-HDL* rather than MDGs. However, the principle is the same. Further translators could be verified and those translators combined into a single verified one.

We now describe how to obtain the new sequential verification linkage theorem. Our combinational linkage theorem is treated similarly. That is, given a linkage theorem about low level code (with semantics `SemProgram_Core`), we use the translator correctnesss theorem to obtain one that converts results from low to high level code (with semantics `SemProgram`). We first instantiate the two machines in terms of the semantics of the *core MDG-HDL* language in the linkage theorem (5). We obtain the linkage theorem based on the semantics of the *core MDG-HDL* language alone as shown below. Note it refers to semantics (`SemProgram_Core`) throughout.

```
⊢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)
```

For any specific verification, we will need to prove the additional assumption about the actual program specification (e.g. SPEC).

$$\forall\ ip.\ \exists\ \ op'.\ SemProgram\_Core(TransProgMC\ SPEC)\ ip\ op' \tag{7}$$

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

We next prove a theorem using the translator correctness theorem (1), *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 ∀ IMP SPEC.
       (∀ 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 linkage 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))
          (SemProgram_Core (TransProgMC IMP))
                       ⊃ (∀ t. flag t = T)) ∧
         (∀ ip. ∃  op'. SemProgram SPEC ip op') ⊃
                       (∀ ip  op. SemProgram IMP ip op ⊃
                         SemProgram SPEC ip op)                 (10)
```

This result, a combination of the translator correctness theorem and linkage theorem allows MDG verification results to be imported into HOL in terms of the semantics of *MDG-HDL*. In the antecedent about the result proved by MDG, it refers to the semantics of the low-level code, `SemProgram_Core`. Its conclusion, however, which is the part corresponding to the result imported into HOL is in terms of the high level code semantics, `SemProgram`.

An example of importing an MDG verification result into HOL is outlined in Section 6.

5.4 Proving the Existential Theorem

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

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

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 output values:

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

Note that similar existential theorems are also needed about implementations: the more normal situation when such a theorem must be proved. 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).

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

This representation might meet an inconsistent model that trivially satisfies any specification. This is sometimes called "The false implies anything problem" [12]. 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 [35], which has the form:

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

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. When importing an MDG result we need also to prove a theorem of this same form for the *specification*, because it is an explicit assumption included in the linkage theorems to allow them to be proved. The stronger consistency theorem (13) is an *existential theorem* for the structural specification, whereas the additional assumption (11) for the linkage theorem is an *existential theorem* for the behavioral specification. We have developed a way of proving both based on the syntax and semantics of the MDG input language [47].

If M represents any specification or implementation of a circuit, and *ip* and *op* to represent the external inputs and outputs, an *existential theorem* will have the form:

$$\vdash_{thm} \forall\ ip.\ \exists\ op.\ M\ ip\ op \tag{14}$$

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

$\vdash_{thm}$ `∀ ip. ∃ op. (∃ op1. SEM_NOT ip op1 ∧ SEM_NOT op1 op)`

We need to strip away the existentially quantified variables of the *existential theorem* then devise and substitute an appropriate *existential term* for each occurrence of an existential variable in the body. As these theorems arise from the syntax of MDG-HDL, the *existential term* of any given variable is determined by one of several *output representations* of the corresponding MDG-HDL components. An *output representation* is just a function giving the component's value as a function in terms of the input and output values at the current or previous time instances. For example, the output representation for a NOT gate is defined as:

```
⊢def existnot (ip:Mdg_Basic) =
        (Bool1_Mdg ( ~ (if ip = BOOL T then T else F)) )
```

where `Bool1_Mdg` is an auxiliary function, which lifts a `boolean` value to a `Mdg_Basic` value. Given an input value at some time instant, the boolean value is extracted, negated and converted back to the lifted version. `existnot` supplied with an appropriate value for the input can then be combined with other output representations for the circuit to create the term used to eliminate an existential quantification over a NOT gate.

In general the following theorem can be used to eliminate existentially quantified variables in a goal if an output representation (`t`) is explicitly represented in the goal.

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

In other words, if the existentially quantified variable (`x`) is explicitly represented by its value as in (15) with (x = t) in the goal, the hidden wire represented by `x` can be removed automatically. In HOL, general purpose simplification tactics can be used to eliminate existentially quantified variables in this way. However, for dealing with those existentially quantified variables which are not represented in the form (x = t), we need to explicitly find their *output representations.*

We have developed a standard method to prove the existential theorems based on the syntax and semantics of *MDG-HDL* [47] [21]. 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 particular design, which reduces the goal `∃ x. t` to `t[u/x]`, is then constructed from the corresponding *output representations*. We have developed HOL tactics both for expanding the semantics of the circuit and proving the *existential theorem*. Further details can be found in [45] and [48].

6 Example: Integrated Hardware and Usability Verification

We have discussed how to prove translator correctness theorems and linkage theorems. Their combination allows MDG verification results to be formalized and reasoned about in HOL in terms of the semantics of *MDG-HDL*. We now consider a simple example, integrating MDG hardware verification and HOL usability verification for a simple vending machine (Figure 8), to illustrate their use. We show how an actual MDG verification result can be imported into HOL to form a traditional HOL theorem that can be used in HOL. Our aim here is not to demonstrate that using the theorems directly in this way is a practical approach – in practice an implementation of the linkage would be used to do that, rather than our theorems which are about the specification of a linked MDG system. The vending machine takes pound coins only, returning change. Lights next to the coin slot and buttons indicate the order things should be done: first a coin is inserted, then the change button is pressed and the change removed, and finally the chocolate button is pressed and the chocolate removed. If the user does not press the appropriate button the machine does nothing until the correct button is pressed. The vending machine, thus, has three inputs corresponding to the buttons being pressed and a coin inserted. It has five outputs: 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 [17]. In the circuit, two registers are needed to store the 4 internal states of the vending machine (reset, coin, choc, change). Their 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.

This example was originally used to verify the absence of a common class of user errors known as post-completion errors within the framework of traditional hardware verification by Curzon and Blandford [18]. A post-completion error is a systematic form of operator error that arises due to human working memory limitations. Users enter interactions with specific goals, such as getting chocolate from a vending machine or a photocopy from a copier. Tasks can be structured in a way that means there are additional tasks that the user must do after achieving the goal, such as taking change or taking back the original being copied.

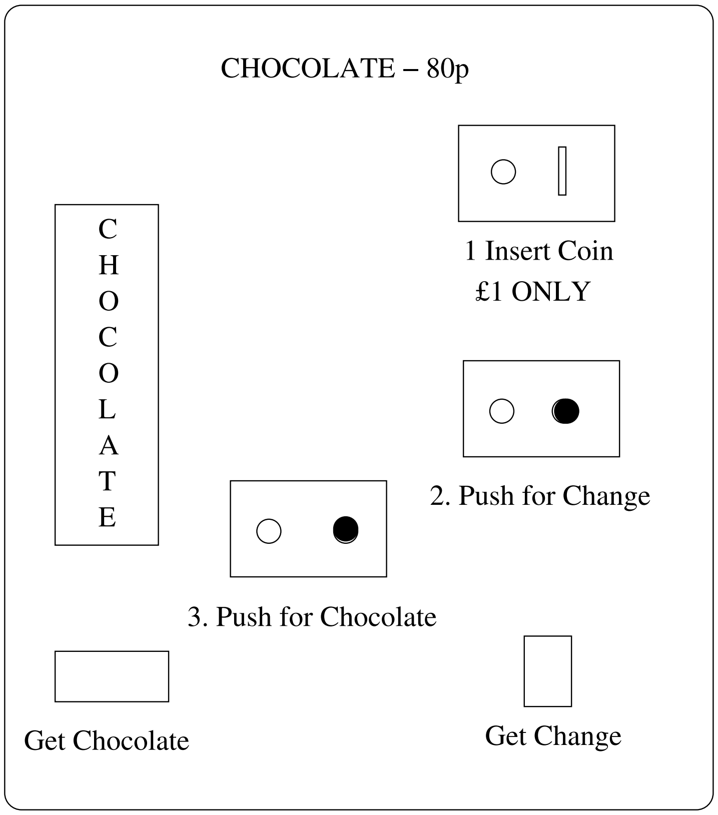

Fig. 8. The Vending Machine

Cognitive psychology studies have shown that such tasks are systematically if unpredictably forgotten [11]. To make this mistake is to make a post-completion error. In the original work of Curzon and Blandford [18], 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 two theorems, the usability theorem based on its *implementation* was proved. In the original work all the verification was done in HOL.

The usability theorems that follow do not include a specific definition of post-completion errors but are more general. What is specified is that the task as a whole (including completion tasks) is eventually completed if users behave in a way as specified by a simple model of user behaviour. If this can be proved then post-completion errors cannot occur (though operator errors for other underlying cognitive reasons may still be possible). The potential of post-completion error is one possible reason for failure.

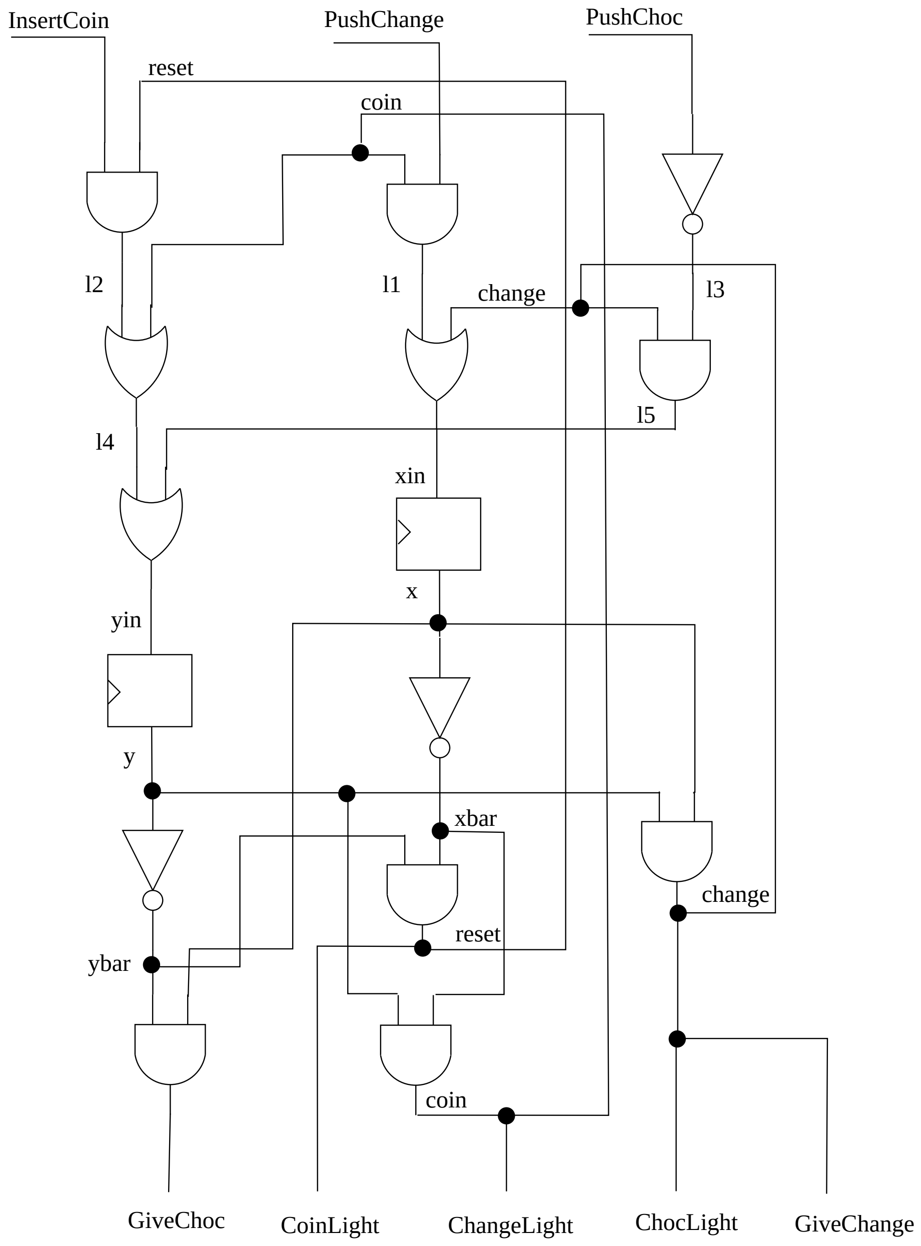

Fig. 9. The Circuit Implementation of the Vending Machine

We closely followed Curzon and Blandford's steps here. However, we have used the MDG system to verify the correctness of the vending machine and imported it into HOL using theorem (10). We then prove the *specification* based usability theorem in the HOL system. By combining those two theorems (the correctness theorem of the vending machine which is verified in MDG and the *specification* based usability theorem which is proved in HOL) we obtain the *implementation* based usability theorem. This demonstrates how an imported theorem (the correctness theorem) can be more than just imported into HOL but can also be used in HOL.

6.1 Verifying the Hardware using MDG

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*.

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

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 linkage theorem for the vending machine can be obtained by instantiating the high level language linkage theorem (10) with the syntax of its implementation and specification (*Vend_Spe_Syn* and *Vend_Imp_Syn*). We obtain theorem *Import_Vend_Thm*:

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

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 linkage theorem (17). This theorem states that the implementation of the vending machine implies its specification.

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

6.2 Usability Verification in HOL

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 [10] based on their goals and knowledge and the physical limitations of human cognition. That is, they specify possible traces of user actions that can be justified in terms of specific results from the cognitive sciences. Of course users might also act outside this behaviour, about which situations the user model says nothing. Its predictive power is bounded by the situations where people act according to the principles specified. All theorems proved using it are thus bounded by that assumption. That does not preclude useful results from being obtained, provided their scope is remembered. The architecture allows us to investigate what happens if a person does act in such plausible ways. If user errors are possible even when such a simple model of behaviour is followed it suggests there is a design problem to be addressed. The behaviour defined by the model is neither "correct" nor "incorrect". It could be either depending on the environment and task in question. It is, rather, "likely" behaviour. We do not model erroneous behaviour explicitly. It emerges from the description of cognitively plausible behaviour.

The user model we use here for demonstration purposes is relatively simple. More complex versions of the model that incorporate a wider range of behaviours have been developed in our more recent work [20]. Here to demonstrate the idea we use the same version used in the original work [18]. In essence the user model consists of a series of non-deterministic guarded rules that give possible next actions of the user. Behavior modeled includes reacting to device prompts, terminating on completing the goal or task, or due to having no other action to take.

This general behaviour is specified as a relation `USER`. Arguments supplied to it give specific available actions, and information about the goals and possessions of the user. In particular, instantiating the model involves specifying concrete types for the machine and user state, a list of pairs of light prompts 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 [18] [19].

```
⊢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 a user-centric property `CHOC_MACHINE_USABLE ustate op ip`. It states that if at any time, `t`, a user approaches the machine when its coin light is on, then they will complete the task as a whole at some time, `t1`, having both chocolate and change: they will thus 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)
```

Our *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: the task will be achieved.

```
⊢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)
```

Here 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.

6.3 Combining the Correctness Results

The *implementation* based usability theorem can be proved in HOL by combining the correctness theorem (19) based on the MDG result with the 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.

$$\begin{aligned}
&\vdash_{thm} \forall\ \texttt{ustate op ip.}\\
&\quad \texttt{Boolean ip op} \wedge\\
&\qquad \texttt{CHOC\_MACHINE\_USER ustate op ip} \wedge\\
&\qquad\qquad \texttt{CHOC\_MACHINE\_IMPL ip op} \supset\\
&\qquad\qquad\qquad \texttt{CHOC\_MACHINE\_USABLE ustate op ip}
\end{aligned} \qquad (21)$$

From this example, we have shown that a system can be verified in two parts by the two different systems and the linkage theorems will justify the way they are combined in the theorem prover. In particular we obtain a theorem of the form obtained if the theorem prover were used alone. 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 linkage theorems giving a greater degree of trust.

The example here is small, and demonstrates using the linkage theorems directly. In a practical situation, the specification of MDG would not be used as that is inefficient. Instead an actual implementation of MDG and the importation process would be used. Confidence in such an implementation would result from it being based on, or verified against, the specification.

7 Conclusions

We have described an approach to formally verifying a linkage between a symbolic state enumeration system and a theorem proving system. Following that of compiler verification work from which this work partially derives, we are concerned with the verification of a specification of the linkage rather than a specific implementation here. The approach involves 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 complete 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 prove 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 approach is to prove general linkage 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. 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 the two systems.

The third step is to combine the translator correctness theorems with concrete versions of the linkage theorems. This combination specifies how verification results from the state enumeration system are 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 justify importing 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 is needed for importing sequential verification results into a theorem proving system. Note that, to import verification results, we have to prove the *existential theorem* for the *behavioral specification* of a design being verified not just of its implementation as this is a required assumption of the linkage theorems. The behavioral specifications must be in the form of a finite state machine or table description. This work makes the linking process easier and removes the burden from the user of the hybrid system.

We have implemented our methodology for a specification of a subset 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 the simplified version of the MDG system. We have provided a formal linkage between the MDG system and the HOL system based on importing theorems [46]. Most importantly, we have combined the translator correctness theorem with the linkage theorems. This justifies not only that MDG verification results can be imported into HOL to form the HOL theorem, but are in a form that can also be used as part of a compositional verification in HOL.

We have described an example verification. We have shown that two different applications (hardware verification and usability verification) suited to two different tools can be combined together. This is not intended to demonstrate the way hardware would actually be verified: that would be done using an implementation of MDG based on the specification. Rather it illustrates the way the linkage theorems work in converting MDG results into HOL ones.

8 Further Work

Due to the limited resources for our project, we have only demonstrated the approach to verifying hybrid verification systems. There are clear ways the work could be taken forward. For example, a linkage based on the full MDG-HDL language used in the Concordia MDG-HOL implementation could be verified. We only verified here the top level translation step. Further translation steps down to MDGs need to be verified for the full system to be verified. In a separate strand of work we verified the translator from *core MDG-HDL* to *MDGs*. This has been integrated with the linkage theorems, for a smaller subset of the language (only using boolean sorts), showing that this is not problematic in principle [45]. We have also not verified the low level algorithms used in the MDG system. If that were done then a verified system stack could be created fully verifying the whole process from verification of result to importation into the theorem prover.

The core MDG algorithms can be used in different ways to implement a variety of different verification applications. We have verified linkage theorems for a selection such as sequential equivalence checking, to illustrate our approach. This could be done for other MDG applications in a similar way.

We have been concerned with verifying the specification. Major work would involve verifying an actual implementation against the specification of the hybrid system. One way this could be done as a simpler alternative to verifying existing implementations would be to develop a new implementation directly from the specification that followed its structure.

We have only considered one linkage – between MDG and HOL – following our methodology for verifying hybrid systems. Important follow up work would be to similarly apply the the approach to other combinations. In preliminary work we verified aspects of a BDD-equivalent subset of MDG so there are no specific issues in that sense of applying the work to BDD based hybrid systems. A specific BDD system could be developed that uses the same approach to product machines such as `PSEQ`. This could be used to illustrate how linkage theorems could be directly applicable to systems based on different low-level enumeration systems. It would also be useful to look at applications based on more complex language translations into MDG: specifications based on industrial languages such as Verilog, for example, rather than the relatively simple MDG-HDL language.

As suggested by an anonymous referee of this paper our approach to proving usability results for the specification and then deducing them for the implementation could also be phrased as a bisimulation problem. An MDG result could be stated as a bisimulation between the implementation and the specification. Once a bisimulation is obtained then any Computational Tree Logic* (CTL*) property that is verified for one model will also hold for the other model. Usability theorems could be stated in CTL*. The state enumeration system could then, for example be used to verify both the usability of the specification and the bisimilarity of the specification and implementation then use HOL to combine the results together to conclude the usability of the implementation. Recasting the work in this way would be very interesting though is beyond the scope of our current project.

Acknowledgments
This work was funded by EPSRC grants GR/M45221, GR/S67494/01 and GR/S67500/01, and a studentship from the School of Computing Science, Middlesex University. Travel funding was provided by the British Council, Canada. We are grateful to the anonymous referees for their valuable comments on early versions of this paper.

References

1. M. D. Aagaard, R. B. Jones, R. Kaivola, and C. J. H. Seger. Formal verification of iterative algorithms in microprocessors. In *Design Automation Conference*, pages 201–206, Washington D.C., USA, June 2000.
2. A. Adams, M. Dunstan, H. Gottliebsen, T. Kelsey, U. Martin, and S. Owre. Computer algebra meets automated theorem proving: Integrating Maple and PVS. In R. Boulton and P. Jackson, editors, *Theorem Proving in Higher Order Logics*, volume 2152 of *Lecture Notes in Computer Science*, pages 27–42, 2001.
3. H. Amjad. Programming a symbolic model checker in a fully expansive theorem prover. In D. Basin and B. Wolff, editors, *Theorem Proving in Higher Order Logics*, volume 2758 of *Lecture Notes in Computer Science*, pages 171–187. Springer-Verlag, 2003.
4. D. Basin, S. Friedrich, and M. Gawkowski. Verified bytecode model checkers. In V. Carreno, C. Munoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 2410 of *Lecture Notes in Computer Science*, pages 47–66. Springer-Verlag, 2002.
5. S. Bensalem, V. Ganesh, Y. Lakhnech, C. Munoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari. An overview of SAL. In C. Michael Holloway, editor, *Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, Virginia, USA, June 2000. NASA Langley Research Center.
6. R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. Van-Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Theorem Provers in Circuit Design*, pages 129–156. North-Holland, 1992.
7. R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions in Computers*, 35(8):677–691, August 1986.
8. R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computer Surveys*, 24(3):293–318, September 1992.
9. B. Buchberger. Towards the automated synthesis of a gröbner bases algorithm. *RACSAM*, 98(1-2):65–75, 2004.
10. R. Butterworth, A.E. Blandford, and D. Duke. Demonstrating the cognitive plausibility of interactive systems. *Formal Aspects of Computing*, 12:237–259, 2000.
11. M. Byrne and S. Bovair. A working memory model of a common procedural error. *Cognitive Science*, 21(1):31–61, 1997.
12. 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, France, September 1986.

13. 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.
14. 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.
15. 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.
16. 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.
17. 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.
18. P. Curzon and A. Blandford. Using a verification system to reason about post-completion errors. In P. Palanque and F. 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.
19. P. Curzon and A. Blandford. Detecting multiple classes of user errors. In M.R. Little and L. Nigay, editors, *Engineering for Human-Computer Interaction*, volume 2254 of *Lecture Notes in Computer Science*, pages 57–71. Springer-Verlag, 2001.
20. P. Curzon, R. Ruksenas, and A. Blandford. Formal verification in human error modelling. Submitted for publication.
21. P. Curzon, S. Tahar, and O. Aït-Mohamed. Verification of the MDG components library in HOL. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher-Order Logics: Emerging Trends*, pages 31–46. Department of Computer Science, The Australian National University, 1998.
22. L. A. Dennis, G. Collins, M. Norrish, R. Boulton, K. Slind, G. Robinson, M. Gordon, and T. Melham. The PROSPER toolkit. In T. Melham and J. Camilleri, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, pages 78–97. Springer Verlag, 2000.
23. 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.
24. 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.
25. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-order Logic.* Cambridge University Press, 1993.
26. O. Grumberg and S. Katz. Veritech: Translating among specification and verification tools design principles. http://www.cs.technion.ac.il/labs/ssdl/research/veritech, March 1999.
27. J. Harrison and L. Théry. A skeptic's approach to combining HOL and Maple. *Journal of Automated Reasoning*, 21:279–294, 1998.
28. 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.

29. A. Heck. *Introduction to MAPLE*. Springer, 1993.
30. J. Hurd. Integrating GANDALF and HOL. Technical Report 461, University of Cambridge, Computer Laboratory, April 1999.
31. J. Joyce and C. Seger. Linking BDD-based symbolic evaluation to interactive theorem-proving. In *Design Automation Conference*, pages 469–474, Dallas, Texas, USA, June 1993.
32. S. Kort, S. Tahar, and P. Curzon. Hierarchical formal verification using a hybrid tool. *Software Tools for Technology Transfer*, 4(3):313–322, May 2003.
33. J Lind-Nielsen. BuDDY - A Binary Decision Diagram Package. http://www.itu.dk/research/buddy/.
34. T. Melham. Integrating model checking and theorem proving in a reflective functional language. In H. Kirchner and C. Ringeissen, editors, *Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, pages 36–39. Springer-Verlag, 2004.
35. T. F. Melham. *Higher Order Logic and Hardware Verification*. Cambridge Tracts in Theoretical Computer Science 31. Cambridge University Press, 1993.
36. T. Mhamdi and S. Tahar. Providing automated verification in HOL using MDGs. In F. Wang, editor, *Automated Technology for Verification and Analysis*, volume 3299 of *Lecture Notes in Computer Science*, pages 278–293. Springer-Verlag, 2004.
37. L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.
38. 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.
39. N. Shankar. Combining theorem proving and model checking through symbolic analysis. In *Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2000.
40. N. Shankar. Using decision procedures with a higher-order logic. In R.J. Boulton and P.B. Jackson, editors, *Theorem Proving in Higher Order Logics*, volume 2152 of *Lecture Notes in Computer Science*, pages 5–26. Springer-Verlag, 2001.
41. S. Tahar, P. Curzon, and J. Lu. Three approaches to hardware verification: HOL, MDG and VIS compared. In G. Gopalakrishnan and P.J. Windley, editors, *Formal Methods in Computer-Aided Design*, volume 1522 of *Lecture Notes in Computer Science*, pages 433–450. Springer-Verlag, 1998.
42. T. Tammet. Gandalf version c-1.0c reference manual. http://www.cs.chalmers.sr/∼tammet/, October 1997.
43. L. Théry. A machine-checked implementation of Buchberger's algorithm. *Journal of Automated Reasoning*, 26(2):107–137, 2001.
44. T.E. Uribe. Combinations of model checking and theorem proving. In H. Kirchner and C. Ringeissen, editors, *Third International Workshop on Frontiers of Combining Systems*, volume 1794 of *Lecture Notes in Computer Science*, pages 151–170. Springer-Verlag, 2000.
45. H. Xiong. *Providing a Formal Linkage between MDG and HOL Based on a Verified MDG System*. Ph.D. thesis, School of Computing Science, Middlesex University, UK, January 2002.
46. 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.

47. 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.
48. H. Xiong, P. Curzon, S. Tahar, and A. Blandford. Proving existential theorems when importing results from MDG to HOL. In R. Boulton and P. Jackson, editors, *TPHOLs 2001 Supplemental Proceedings*, Informatic Research Report EDI-INF-RR-0046, pages 384–399, September 2001.
49. Z. Zhou and N. Boulerice. *MDG Tools (V1.0) User Manual.* University of Montreal, Dept. D'IRO, 1996.
50. Z. Zhou, X. Song, S. Tahar, E. Cerny, F. Corella, and M. Langevin. Verification of the island tunnel controller using multiway decision graphs. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 233–246. Springer-Verlag, 1996.

Appendix: The Syntax of the MDG-HDL Language

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

Default_Type ::= DENORMAL of Mdg_Basic |
                DEOUT of out_type |

Table_Val ::= TABLE_VAL of α | DON'T_CARE

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

Mdg_Hdl ::= NOT of string=>string |
          AND of string=>string=>string |
          OR of string=>string=>string |
          NAND of string=>string=>string |
          XOR of string=>strin=>string |
          NOR of string=>string=>string |
          AND3 of string=>string=>string=>string |
          OR3 of string=>string=>string=>string |
          NAND3 of string=>string=>string=>string |
          NOR3 of string=>string=>string=>string |
          AND4 of string=>string=>string=>string=>string |
          OR4 of string=>string=>string=>string=>string |
          NAND4 of string=>string=>string=>string=>string |
          NOR4 of string=>string=>string=>string=>string |
          AND5 of string=>string=>string=>string=>string=>string |
          OR5 of string=>string=>string=>string=>string=>string |
          NAND5 of string=>string=>string=>string=>string=>string |
          NOR5 of string=>string=>string=>string=>string=>string |
          AND6 of string=>string=>string=>string=>string=>string=>string |
          OR6 of string=>string=>string=>string=>string=>string=>string |
          NAND6 of string=>string=>string=>string=>string=>string=>string |
          NOR6 of string=>string=>string=>string=>string=>string=>string |
          JKFF of string=>string=>string |
          RSFF of string=>string=>string |
          JKFFE of string=>string=>string=>string |
          AO of string=>string=>string=>string=>string |
          REGCON of string=>string=>string |
          REG of string=>string |
          FORK of string=>string |
          INIT of (string#Mdg_Basic) |
          SNXT of string=>string |
          TABLESYN of (string list)=>Out_Type=>((Mdg_Basic Table_Val list) list)
                       =>(Mdg_Basic list)=> Default_Type |
          JOIN of Mdg_Hdl=>Mdg_Hdl |
          INTERNAL of string => Mdg_Hdl

Exoutput ::= EXOUT of string list
Exinput ::= EXIN of string list
Invariable ::= INV of string list
Mdg_Program ::= PROG of Exoutput =>Exinput => Invariable => Mdg_Hdl
```