Hierarchical Verification Using an MDG-HOL Hybrid Tool

Iskander Kort[1], Sofiene Tahar[1] and Paul Curzon[2]

[1]Concordia University, Canada. ({tahar,kort}@ece.concordia.ca)
[2]Middlesex University, UK. (p.curzon@mdx.ac.uk)

Abstract. We describe a hybrid formal hardware verification tool that links the HOL interactive proof system and the MDG automated hardware verification tool. It supports a hierarchical verification approach that mirrors the hierarchical structure of designs. We obtain advantages of both verification paradigms. We illustrate its use by considering a component of a communications chip. Verification with the hybrid tool is significantly faster and more tractable than using either tool alone.

1 Introduction

Automated decision diagram based formal hardware verification is fast and convenient, but does not scale well, especially where data paths and control circuitry are combined. Details of the version of the design verified need to be simplified: e.g., considering 1-bit instead of 32-bit datapaths. Finding a model reduction and appropriate abstractions so that verification is tractable with the tool can be time-consuming. Moreover, significant detail can be lost. An alternative is interactive theorem proving. The verification can be done hierarchically allowing large designs to be verified without simplification. Furthermore it is possible to reason about high level abstractions of datatypes. It can however be very time-consuming, requiring significant user interaction and skill.

The contribution of our work is to implement a hybrid tool combining HOL [9] and MDG [4] which provides explicit support for hierarchical hardware verification. In particular, we have provided an embedding of the MDG input language in HOL, implemented a linkage between HOL and MDG using the PROSPER toolkit [7] and implemented a series of HOL tactics that automate hierarchical verification. This means that a hierarchical proof can be performed as it might be done using a pure HOL system. However, the MDG tools can be seemlessly called to perform verification of components that are within its capabilities. We have verified a component of a communication switch using the tool. Verification is shown to be significantly faster and more tractable using the hybrid tool than with either tool individually.

The remainder of this paper is organized as follows. In Section 2 we overview briefly the two tools being linked. We present our hybrid tool and the methodology it embodies in Section 3. A case study using the tool to verify a component of an ATM switch is described in Section 4. Finally, we discuss related work in Section 5 and draw conclusions in Section 6.

2 The Linked Tools

Our hybrid tool links the HOL interactive theorem prover and the MDG hardware verification system. HOL [9] is based on higher-order logic. The user works interactively with the system calling SML functions that implement inference rules to apply proof steps. New theorems are created in HOL by applying inference rules—derived rules call a succession of primitive rules, thus the user can have great confidence in the derived theorems. However, HOL also provides functions to create theorems directly without proof. This feature can be used to import results produced by external tools into HOL. We initially used the PROSPER/Harness Plug-in Interface of HOL [7]. This gives a uniform way of linking HOL with external proof tools. It provides the low level client-server communication interface from HOL to various languages within which other tools are integrated. A range of different external proof tools (such as MDG) can act as servers to a HOL client. The interface removes the burden of writing low-level communication tools, leaving the hybrid tool designer to concentrate on higher-level issues. It also tags theorems produced by plug-ins with a label indicating their source. These labels are propagated to any theorem generated from the imported result allowing the pedigree of any result to be later determined.

The MDG system, which is primarily designed for hardware verification, provides verification procedures for equivalence and property checking. The former provides the verification of two combinational circuits or the verification of two state machines. The latter allows verification through invariant checking or model checking. The strength of the MDG system is its automation and ease of use. It has been used in the verification of significant hardware examples [3, 16, 18]. The MDG system is a decision diagram based verification tool based on Multiway Decision Graphs (MDGs) [4] rather than on BDDs. MDGs overcome the data width problem of Reduced-Order Binary Decision Diagram (ROBDD) based verification tools. An MDG is a finite, directed acyclic graph (DAG). MDGs essentially represent relations rather than functions. They are much more compact than ROBDDs for designs containing a datapath. Furthermore, sequential circuits can be verified independently of the width of the datapath. The MDG tools combine some of the advantages of representing a circuit at more abstract levels with the automation offered by decision-diagram based tools. The input language for MDG, MDG-HDL, supports structural descriptions, behavioral descriptions as Abstract State Machine (ASM) or a mixture of both. A structural description is usually a netlist of components connected by signals, and a behavioral description is given by a tabular representation of the transition/output relation of the ASM. This is done using the Table construct of MDG-HDL: essentially a case statement that allows the value of a variable to be specified in terms of the values of inputs and other expressions

3 The Hybrid Tool and Verification Methodology

In a pure MDG verification, structural and behavioral descriptions are given for the top level design. An automated verification procedure is then applied. If the problem is sufficiently tractable, the verification is completed automatically. If not, ideally the problem would be attacked in a hierarchical fashion by verifying the sub-blocks independently. However, the management of this process cannot be done within the tool, though could be done informally outside it.

In a pure HOL hardware verification, the proof is structured according to the design hierarchy of sub-blocks within the implementation. For each block, including the top level block of the design, a structural specification and behavioral specification are given. Each block's implementation (apart from those at the bottom of the hierarchy) is verified against its specification in three steps. Firstly an intermediate verification result is obtained about the block based on the behavioral descriptions of its sub-blocks. Essentially the sub-blocks are treated as primitive components in this verification. Secondly the process is repeated recursively on the sub-blocks to obtain correctness theorems for them. Finally, the correctness theorems of the sub-blocks are combined with the intermediate correctness theorem of the block itself to give the actual correctness theorem of the block. This is based on the full structural description of the block down to primitive components. The verification follows the natural design hierarchy. If this process is applied to the top level design block, a correctness theorem for the whole design is obtained. The integration of the verification results of the separate components that would be done informally (if at all) in an MDG verification is thus formalized and machine-checked in the HOL approach.

Our hybrid tool supports hierarchical verification, automating the process discussed above, and fits the use of MDG verification naturally within the HOL framework of compositional hierarchical verification. The HOL system is used to manage the proof, with the MDG system called seemlessly to verify those design blocks that are tractable. This removes the need to provide behavioral specifications for sub-blocks and the need to verify them separately. In particular, if the design of any sub-block is sufficiently simple, then the hierarchical approach can be abandoned for that block and the whole block verified in one go in MDG. Furthermore, verifying a block under the assumption that its sub-blocks are all primitive components may also be done using MDG if tractable. If not, a normal HOL proof can still be performed. No information is lost in using MDG via the hybrid tool. To allow the seamless integration of the tools, we use MDG-style behavioral specifications within HOL. This means the specifications must be in the form of a finite state machine or table description. If a higher level abstraction, unavailable in MDG, is required then a separate HOL proof is performed that an MDG style specification meets this abstraction.

3.1 The Hybrid Tool

Our Hybrid tool was written in SML. It consists of five modules: a parsing module, an extraction module, a hierarchical verification support module, a code generation module and an MDG interaction module (cf. Figure 1). Subgoal management is done using the HOL subgoal manager. This is an advantage of the hybrid approach—the existing HOL infrastructure augments MDG providing a much more powerful interface to MDG.

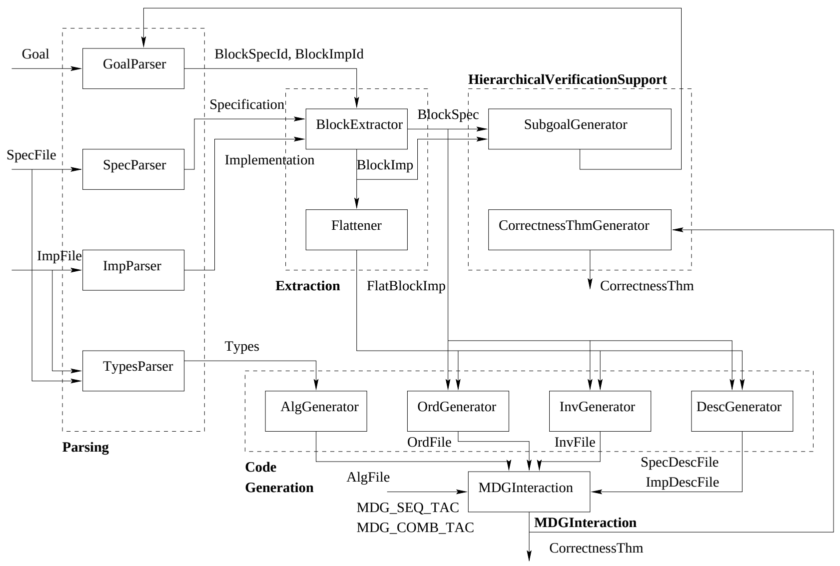

Fig. 1. The Hybrid Tool's Structure.

The hybrid tool supports the hierarchical verification process by providing a HOL embedding of the concrete subset of the MDG input language to allow MDG-style specifications to be written in HOL. Three high-level proof tactics that manage the proof process are also provided. A hierarchy tactic, HIER_VERIF_TAC, automates the subgoaling of the correctness theorem of a block by analyzing its structure as outlined in the previous section. It later combines the proven subgoals to give the desired correctness theorem. Where a non-primitive component occurs several times within a block, the tactic avoids duplication, generating a single subgoal that once proved is automatically instantiated for each occurence of that component to prove the correctness of the block. Two other tactics automate the link to the MDG tools: MDG_COMB_TAC attempts to verify a given correctness theorem for a block using MDG combinational equivalence verification; MDG_SEQ_TAC calls MDG sequential equivalence verification to prove the result.

Verification using the hybrid tool proceeds as shown in Figure 2. An initial goal is set that the top level design's implementation meets its behavioral specification. If the design can be verified using MDG, the appropriate MDG tactic, determined by whether the circuit is sequential, is called. Otherwise, the hierarchy tactic is called to break the design into smaller parts, and the process is repeated. At any point, a HOL proof can be performed directly to prove a goal. MDG verification can fail due to state-space explosion leading to the system running out of memory. In general MDG can fail to terminate, however the current version of the hybrid tool does not do so due to the fact that abstract variables are not yet supported.

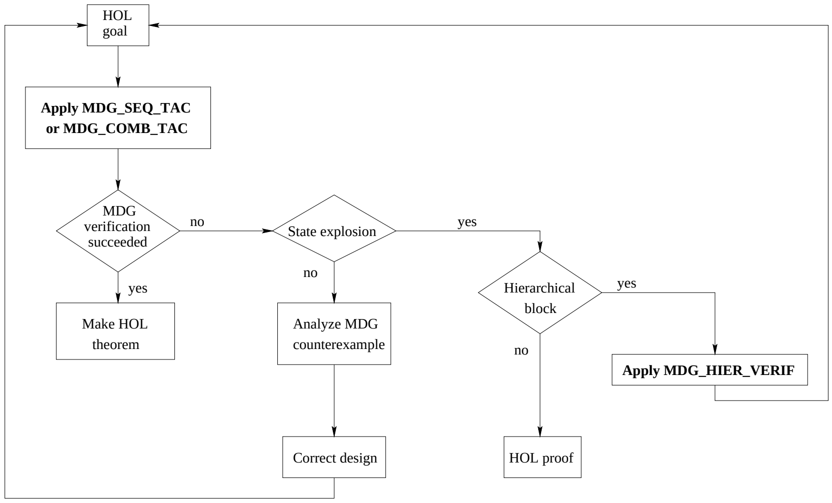

Fig. 2. Using the Hybrid Tool

3.2 Specifications

The hybrid tool must be supplied with a behavioral specification for each block in the design that is verified. This is not necessary for sub-blocks within blocks verified by calls to MDG. The specifications are provided as a normal file of HOL definitions. However, as these definitions must be analyzed by the tool and ultimately converted into MDG, they must follow a prescribed form: they must consist of a conjunction of tables, which input and output arguments must both be explicitly typed and be in a given order. MDG abstract variables are not currently supported. The tables are an embedding of MDG tables in HOL originally defined by Curzon *et. al.* [5] to verify the MDG components in HOL. The verification of these components increases confidence that the MDG tools can be trusted when used in the HOL system.

Structural specifications are written in a subset of the HOL logic similar to that for behavioral specifications. However, the descriptions are not limited to tables but can include any component of the MDG component library. The structural specification of a block thus differs from a behavioral specification in that its body consists of a network of components. A component may be an MDG built-in component, a functional block, a table or a component previously defined by the user. The MDG built-in components are an embedding in HOL of the actual MDG components.

3.3 The Verification Process

The hybrid tool is intended to provide automated support for hierarchical verification and to enable the user to verify some blocks using MDG. We will illustrate this by refering to the verification of a simple adder circuit. A typical session

HA_i ((x:num $\rightarrow$ bool), (y:num $\rightarrow$ bool)) ((z:num $\rightarrow$ bool), (cout:num $\rightarrow$ bool)) =
(MDG_XOR (x,y) z) $\wedge$ (MDG_AND (x,y) cout)

FA_i ((x:num $\rightarrow$ bool), (y:num $\rightarrow$ bool),(cin:num $\rightarrow$ bool))
((z:num $\rightarrow$ bool), (cout:num $\rightarrow$ bool)) =
$\exists$ (z_0:num $\rightarrow$ bool) ($cout_0$:num $\rightarrow$ bool) ($cout_1$:num $\rightarrow$ bool).
(HA_i (x,y) (z_0,$cout_0$)) $\wedge$ (HA_i (z_0,cin) (z,$cout_1$)) $\wedge$
(MDG_OR ($cout_0$,$cout_1$) cout)

Fig. 3. A Structural Specification of an Adder

z_TAB ((x:num $\rightarrow$ bool), (y:num $\rightarrow$ bool)) (z:num $\rightarrow$ bool) =
TABLE [x;y] z [[F; F]; [T; T]] [F;F] T

cout_TAB ((x:num $\rightarrow$ bool), (y:num $\rightarrow$ bool)) (cout:num $\rightarrow$ bool) =
TABLE [x;y] cout [[F; DONT_CARE]; [T; F]] [F;F] T

HA ((x:num $\rightarrow$ bool), (y :num $\rightarrow$ bool)) ((z:num $\rightarrow$ bool), (cout :num $\rightarrow$ bool)) =
(z_TAB (x,y) z) $\wedge$ (cout_TAB (x,y) cout)

Fig. 4. A Behavioural Specification of a Half-Adder

with the hybrid tool goes through the following steps. First, the user supplies the tool with a specification file and an implementation file as part of an initialization procedure. These are SML files containing normal SML definitions. The specification file includes the behavioral specifications of the design blocks. The implementation file includes the design structural specification and follows the design hierarchy. Both files may include user defined HOL datatypes. An example of a structural specification for an adder is given in Figure 3. The behavioral specification of a half-adder in terms of tables is given in Figure 4. The specification of the full adder is similar. In a table specification, the first list gives the inputs of the table, the next argument is the output. Next is a list of lists giving possible combinations of input values and then a list giving the output values resulting from those combinations. The final argument gives the default value for any combination of inputs not listed. MDG tables are more general than shown in this example in that general expressions can be used as table inputs and variables can appear in the rows. For example, the carry out signal in the half-adder is defined by a table with two inputs x and y and one output cout. If x is False and y is “DON'T CARE” (i.e. anything) then cout is False. Similarly if x is true and y is false then cout is false. The default value for all other combinations of input values is true. The behavioral specifications of the components are similarly defined. The initialization procedure also involves loading the embeddings of the MDG tables and the MDG components in HOL as well as starting a server to the MDG system.

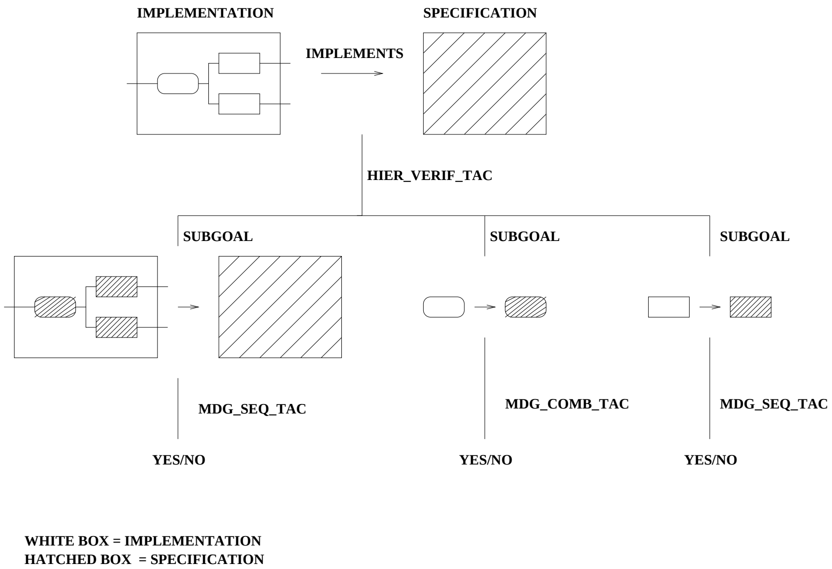

Fig. 5. Hierarchical Verification using HIER_VERIF_TAC.

Once the tool is initialized, the user sets the correctness goal for the whole design using HOL's subgoal package. This goal states that the design's implementation implies its specification. For example, for our adder, we set the goal:

∀ x y cin z cout. FA_i (x,y,cin) (z,cout) $\Longrightarrow$ FA (x,y,cin) (z,cout)

This correctness goal could then be resolved directly through MDG using MDG_SEQ_TAC or MDG_COMB_TAC. Applying these tactics to complex designs may lead to state explosion. To overcome this, HIER_VERIF_TAC is used. The action of this tactic is summarized in Figure 5. It automatically generates a correctness subgoal for every immediate sub-block in the design. Where one sub-block is used in several places, only one goal is generated: the hybrid tool generates a general subgoal that justifies its use in each situation. A further subgoal states that the lower level specifications, connected according to the structural specification, imply the current specification.

For example, HIER_VERIF_TAC generates two subgoals for our adder.

∀ x y z cout. HA_i (x,y) (z,cout) $\Longrightarrow$ HA (x,y) (z,cout)

∀ x y cin z cout. FA_i_hl (x,y,cin) (z,cout) $\Longrightarrow$ FA (x,y,cin) (z,cout)

The first is a correctness statement for the half-adder component. Only one general version is generated. This is used to create the two theorems justifying each of the two instances of this component in the design. The second subgoal is a correctness goal for the adder where the half-adder is treated as a primitive component. It contains an automatically generated new structural specification

FA_i_hl, which is in terms of the behavioral specifications of the half-adder submodules rather than their structural specifications:

$$
\begin{array}{l}
\vdash \text{FA\_i\_hl } ((\text{x:num} \rightarrow \text{bool}), (\text{y:num} \rightarrow \text{bool}),(\text{cin:num} \rightarrow \text{bool})) \\
\qquad\qquad ((\text{z:num} \rightarrow \text{bool}), (\text{cout:num} \rightarrow \text{bool})) = \\
\qquad\qquad\quad \exists\ (\text{z}_0\text{:num} \rightarrow \text{bool})\ (\text{cout}_0\text{:num} \rightarrow \text{bool})\ (\text{cout}_1\ \text{:num} \rightarrow \text{bool}). \\
\qquad\qquad\qquad\qquad (\text{HA (x,y) (z}_0\text{,cout}_0)) \wedge (\text{HA (z}_0\text{,cin) (z,cout}_1)) \wedge \\
\qquad\qquad\qquad\qquad (\text{MDG\_OR (cout}_0\text{,cout}_1\text{) cout})
\end{array}
$$

HIER_VERIF_TAC creates a justification function that given theorems corresponding to the subgoals creates the theorem corresponding to the original goal. The subgoals it produces could be resolved using a conventional HOL proof, by invoking MDG as above or by applying HIER_VERIF_TAC once again. If the subgoals are proved, then the justification rule of HIER_VERIF_TAC will automatically derive the original correctness goal from them. In our example, we apply one of the MDG-based tactics. This circuit is purely combinational so MDG_COMB_TAC is used.

When the MDG-based tactics are applied, the hierarchy in the structural specification is automatically flattened to the non-hierarchical form of primitive components required by MDG (just the next layer down in the case of the second subgoal above). The tool currently generates a static variable ordering for use by MDG though more sophisticated ordering heuristics could be included. Alternatively the tool user can provide the ordering. Each block verified can use a different variable ordering.

The tool analyses the feedback of MDG in order to find out whether the verification succeeded or failed. If the verification fails a counter-example is generated. If it succeeds, the tactic creates the appropriate HOL theorem. For example, for our adder we obtain the theorems:

$$\text{[MDG]} \vdash \forall\ \text{x y z cout. HA\_i (x,y) (z,cout)} \Longrightarrow \text{HA (x,y) (z,cout)}$$

$$\text{[MDG]} \vdash \forall\ \text{x y cin z cout. FA\_i\_hl (x,y,cin) (z,cout)} \Longrightarrow \text{FA (x,y,cin) (z,cout)}$$

The theorem is tagged with an oracle label indicating that it is proved by an external tool. This tag will be passed to any theorem proved using these theorems.

Note also that the theorem proved can be instantiated for any instance. We effectively can prove a single correctness theorem for a block and reuse it for any instance of the block. In our example, there are two instances of the half-adder, but this single theorem is used for both. This process is managed formally and machine-checked within HOL. This contrasts with pure automated tools, where each instance would need a specific theorem to be verified separately or non-machine-checked reasoning to be relied upon. For the half-adder, the subgoals are formally combined using automatic proof by HIER_VERIF_TAC to give the desired theorem about the adder:

$$\text{[MDG]} \vdash \forall\ \text{x y cin z cout. FA\_i (x,y,cin) (z,cout)} \Longrightarrow \text{FA (x,y,cin) (z,cout)}$$

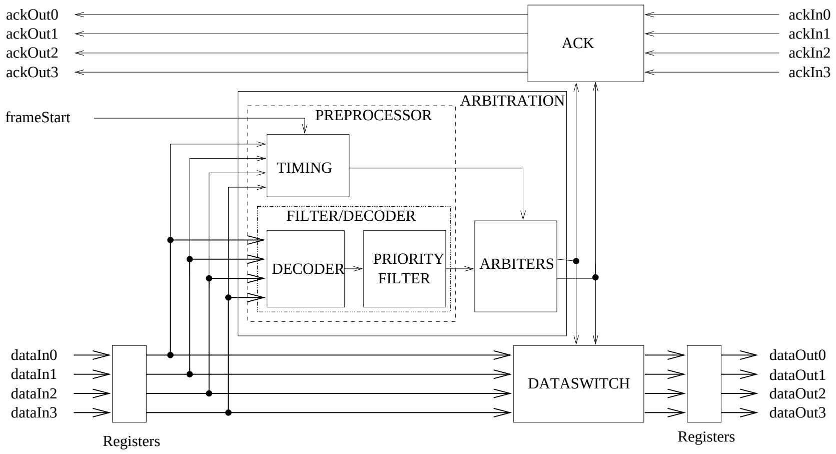

Fig. 6. The Fairisle ATM Switch Fabric.

The way HOL and MDG are used together is thus that the former manages the compositional aspects of the proof, ensuring duplicated work is avoided. The latter does fast, automated, low-level verification.

4 Case Study: The 4×4 ATM Switch Fabric

We have applied the hybrid tool to a realistic example: the verification of a block of the Fairisle ATM (Asynchronous Transfer Mode) switch fabric [13]. The Fairisle switch fabric is a real switch fabric designed and used at the University of Cambridge for multimedia applications. It switches cells of data from 4 input ports to 4 output ports as requested by information in header bytes in each cell.

Curzon [6] formally verified this ATM switching element hierarchically using the pure HOL system. However, this verification was very time-consuming. Verifying the fabric can be done hierarchically following exactly the same structure as the original design using our hybrid tool. However, with the tool, many of the sub-blocks can be verified automatically using the MDG tool, thus saving a great deal of time and effort. Furthermore, HIER_VERIF_TAC automates much of the management of the proof that was previously done manually. Attempting the verification in MDG alone would, on the other hand, be barely tractable taking days of CPU time. This is discussed in more detail below.

The fabric is split into three sub-blocks, namely Acknowledgement, Arbitration and Data Switch. Further dividing the Arbitration sub-module, we have essentially two blocks: the arbiters that make arbitration decisions and a preprocessing block that generates the timing signal and processes the headers of the cells into a form usable by the arbiters (see Figure 6). We consider the verification of the preprocessor block here (see Figure 7). The timing block within the preprocessor generates a timing signal for the arbiters from an external frame signal

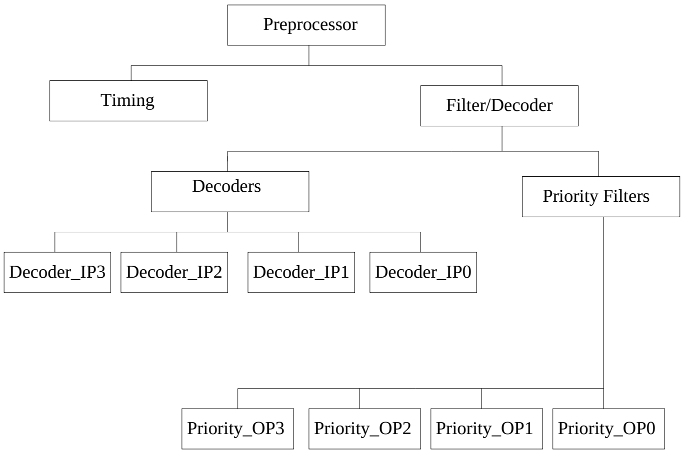

Fig. 7. The Preprocessor Hierarchy.

and from the data stream. The decoder block (made of 4 independent decoders) takes the four cell headers from the data stream and extracts the information about the destinations they are requesting (which is in a binary encoding). For each destination a unary encoding of the cells that are requesting that output is created. The priority filter takes this information together with priority information from the cell headers. If any cell has high priority, then requests from low priority cells are not forwarded to the arbiters.

Setting as goal the correctness statement for the preprocessor, we attack it using HIER_VERIF_TAC. We obtain two subgoals corresponding to the timing block and the filter-decoder block, together with a subgoal that the combined preprocessor is correct on the assumption that its sub-blocks are. As the Timing block is a sequential design, we call MDG_SEQ_TAC to automatically prove the timing unit correctness subgoal. This proves the equivalence of the implementation and its specification, and so proves the implication in our subgoal.

Decoders and *Priority Filters* are purely combinational circuits. Their specifications are the conjunctions of 32 16-input-tables and 16 32-input-tables, respectively. MDG takes 16 hours to verify *Decoders* and it would take days to verify *Priority Filters*. The problem is in finding an efficient variable ordering given that the way the sub-blocks are connected means that the best ordering for one table is bad for another. In order to overcome this problem, we move down one level in the design hierarchy. More specifically, the 32 tables in *Decoders*' specification were partitioned into four 8-table-sub-blocks: *Decoder_IP0* ... *Decoder_IP3*. *Decoder_IPi* is a decoder for input port i, $i = 0..3$. A more efficient variable ordering is then supplied for each of these sub-blocks. Similarly, the 16 tables in *Priority Filters*' specification were partitioned into four 4-table-sub-blocks: *Priority_OP0* ... *Priority_OP3*. *Priority_OPi* is a priority filter for output port i, $i = 0..3$. The preprocessor hierarchy as verified is shown in Figure 7.

Block	CPU Time (sec.)
Preprocessor	495.230
Timing	0.060
Filter/Decoder	488.900
Decoders	45.520
Decoder_IPi	10.050
Priority	437.210
Priority_OPi	107.413

Table 1. Hierarchical Verification Statistics.

We apply HIER_VERIF_TAC to verify *Decoders* and *Priority Filters* based on this hierarchy. The subgoals associated to *Decoder_IPi* and *Priority_OPi*, $i = 0..3$, are then proved automatically. Note that this still avoids expanding the hierarchy as far as in the original HOL proof—so lower level behavioral specifications do not need to be written.

Table 1 shows the hierarchical verification statistics, including CPU time in seconds. Obviously, using our hybrid tool, the verification of the preprocessor is faster than proving in HOL that the implementation implies the high-level specification. Given the formal specifications, Curzon [6] originally took several days to do the proofs of these blocks using interactive proof whereas the verification is done in minutes using our tool. Verification is also faster than using MDG alone: splitting the decoder block enabled verifying it within less than 1 minute using our hybrid tool instead of 16 hours if only MDG was used. It took a day (approximately 8 hours) to interactively prove the decoder block in HOL. Thus verification is faster using the hybrid tool than with either system on its own as shown in Table 2 which gives approximate times for verifying the decoder block. These times should be treated with caution, as the pure HOL times are not CPU time but that for the human to interactively manage the verification. Times to develop specifications, including those of sub-blocks verified hierarchically rather than directly using MDG, are not included in these times. Though, writing these specifications was straightforward. It therefore is worthwhile additional work, given the overall time improvement. Some extra human interaction time for the verification part is also needed when using the hybrid tool over the bare CPU time. This is needed to call the appropriate tactics. However, this is minimal—a matter of minutes rather than hours, since it follows the existing design hierarchy. The main part that is time consuming is if unsuccessful automated proofs of sub-blocks are attempted. This obviously requires judgement over the limitations of MDG, in knowing when it is worth attempting automated proof, and when it is better to step down a level in the hierarchy.

5 Related Work

Work to combine the advantages of automated and interactive tools falls generally into two areas: hybrid tools in which two existing, stand-alone verification

HOL (Human Proof Time)	MDG (CPU Time)	Hybrid Tool (CPU Time)
Interactive	Automated	Semi-automated
8 hours	16 hours	1 minute

Table 2. Comparison of Verifications of the Decoder Blocks

systems are linked; and systems where external proof packages are embedded as decision procedures for some subset of the logic by an interactive system.

Perhaps the most impressive hybrid verification system to date is the combined Voss-ThmTac System [2]. It combines a simple, specially written LCF style proof system, ThmTac with the Voss Symbolic Trajectory Analysis System. This system evolved out of the HOL-VOSS System [11]. In that system, Voss was interfaced within HOL as a tactic that could be called to perform a symbolic trajectory analysis to verify assertions about sequences of states. The Voss-ThmTac System is thus based on many years of experience combining systems. It has been used to verify a series of real hardware designs including an IA-32 Instruction length decoder claimed to be one of the most complex hardware verifications completed. Much of its power comes from the very tight integration of the two provers allowing the user to interact directly with either tool. This is facilitated by the use of a single language, *fl*, as both the theorem prover's meta-language and its object language.

Schneider and Hoffmann [15] linked SMV (a CTL model checker) to HOL using PROSPER. In this hybrid tool, HOL conversions were used to transform LTL specifications into ω-Automata, a form that can be reasoned about within SMV. These HOL terms are exported to SMV through the PROSPER plug-in interface. On successful model checking the results are returned to HOL and turned into tagged theorems. This allows SMV to be used as a HOL decision procedure. The SMV specification language has also been deeply embedded in HOL, allowing temporal logic specifications to be manipulated in HOL and the model checker used to return a result about its validity.

The use of tightly integrated decision procedures is a major focus of the PVS proof system. Rajan *et al* [14] integrated a BDD-based model checker for the propositional μ-calculus within PVS. An extension of the μ-calculus is defined within higher-order logic and temporal operators then defined as μ-calculus fixpoint definitions. These expressions are converted into the form required by the model checker which can then be used to prove appropriate subgoals generated within PVS. Such results are treated no differently to those created by proof.

An issue with accepting imported results as theorems is whether the external system can be trusted to produce "theorems" that really are host system theorems. This is more of an issue with fully-expansive proof systems such as HOL where the integrity of the system depends on a small core of primitive inference rules. Accepting results from an external package essentially treats that package as one of the trusted primitives. The approach taken by Gordon [8] to minimize this problem in the BuDDy package when integrating BDD based tools is to provide a small set of BDD primitives in terms of which full tools are implemented. In this way only the primitives need to be trusted not the whole package.

Hurd [10] used PROSPER to combine the Gandalf prover with HOL. Unlike other approaches, the system reproves the Gandalf theorems within HOL rather than just accepting the results. The Gandalf proof script is imported into the HOL system and used to develop a fast proof within HOL. The tool is thus used to discover proofs, rather than directly to prove theorems.

The MEPHISTO system [12] was developed to manage the higher levels of a verification, producing first-order subgoals to be proved by the FAUST first order prover. The goals of MEPHISTO are similar to ours: managing the subgoaling of a verification to produce goals that can be proved by another system. The difference is the focus of the way the systems do this and the target system. Our approach is to use the existing design hierarchy, sending to the automated prover (here a hardware verification system itself) subgoals that are correctness theorems about design modules. Thus HIER_VERIF_TAC produces subgoals (and results from failed verification) easily understood by the designer. This approach avoids the problem of the verifier having to inspect goals that bear little relation to the input to the system. MEPHISTO does give some support for hierarchical proof providing a library of preproved modules. However, in our approach such hierarchical verification is explicitly supported by the tactics.

Aagaard *et al* [1] proposed a similar hardware verification management system. They aimed to complete the whole proof within the theorem prover (HOL or Nuprl). As with MEPHISTO, the focus is on producing lemmas to be proved by decision procedures. They developed a series of prototype tactics that could be used to break down subgoals. However, they do not directly support hierarchical verification: the first step proposed is to rewrite with the module specifications.

As in [2] and [15], we integrate a theorem prover (HOL) to an existing hardware verification tool (MDG) rather than embedding a package within the system. We work within the proof system but using the specification style of the automated tool. This is done by embedding the language of the automated verification tool within the proof system. As is done in pure HOL verification, the proof follows the natural design hierarchy embodied in the specifications. This process is explicitly supported by our hierarchy tactic. By working in this way we obtain a seamless integration of the tools. The subgoals automatically generated also have a direct relation to the specifications produced by the designer.

6 Conclusions

We have described a tool linking an interactive theorem prover and an automated decision diagram-based hardware verification system. This builds on previous work [17], where we showed formally how an MDG equivalence proof can be imported to an implication-based correctness theorem in HOL. Our system explicitly supports the hierarchical compositional verification approach naturally used in interactive proof systems, when using an automated tool. The interactive proof system is used to automatically manage the proof as well as complete any proof interactively that is beyond the scope of the automated system. The verification of whole blocks in the hierarchy can however be done automatically. The hybrid tool can be used to verify larger examples than could be done in MDG alone, and these proofs can be done faster than in either system alone.

We used the PROSPER toolkit to perform the linkage of the two tools. This made providing such a linkage relatively easy. However, with the early version of PROSPER used the linkage was slow. An alternative implementation that communicated between the tools directly using files was quicker.

We illustrated the use of the hybrid tool by describing the verification of the preprocessing block of the arbitration unit of an ATM switch. This was done using hierarchical verification with both the combinational and sequential equivalence checking tools of MDG being used. Using the hybrid tool, a verification that originally required many hours of interactive proof work, could be done largely automatically using the hybrid tool.

We intend to extend the capabilities of the tool to increase the automation of the proof management process. For example, we will automate different forms of parameterization. Parameterized circuits must currently be dealt with interactively. A single instance of the parameterized circuit is verified using the hybrid tactics and this theorem used in a pure HOL proof of the parameterized circuit—performing the inductive part of the proof. This process could be automated for a range of common parameterization patterns (see Aagaard *et al* [1]) with a similar tactic to HIER_VERIF_TAC managing the inductive part of the proof. Common abstraction techniques to reduce a model say from 32-bits to 1 bit to make automated verification tractable could also be dealt with in this way. However, MDG provides a better approach: by making fuller use of the abstraction facilities in MDG itself we will remove the need for such abstraction. Currently only bit level designs can be verified using MDG via the hybrid tool. However, a future version of the hybrid tool will allow designs with abstract variables to be exported. This will remove the need to simplify datapath widths to make verification tractable and will enable us to handle data-dependent circuits automatically. We will also extend the hybrid tool to support model checking in MDG. While most of the infrastruture may be reused, ways of translating and composing temporal properties in HOL need to be developed. Finally, we will consider the verification of more complex examples including a full 16 by 16 switch fabric.

Acknowledgments This work was funded by the NSERC Strategic Grant STP0201836 and EPSRC Research Agreement GR/M45221.

References

1. M.D. Aagaard, M. Leeser, and P. Windley. Toward a super duper hardware tactic. In J.J. Joyce and C.H. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications*, LNCS 780, pages 400–413. Springer-Verlag, 1993.

2. M.D. Aagaard, R.B. Jones, and C-J.H. Seger. Lifted-FL:A Pragmatic Implementation of Combined Model Checking and Theorem Proving. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics*, LNCS 1690, pages 323–340. Springer-Verlag, 1999.
3. S. Balakrishnan and S. Tahar. A Hierarchical Approach to the Formal Verification of Embedded Systems Using MDGs. In *Proceedings IEEE 9th Great Lakes Symposium on VLSI*, Ann Arbor, Michigan, USA, March 1999, pages 284–287.
4. F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny. Multiway Decision Graphs for Automated Hardware Verification. *Formal Methods in System Design*, 10(1):7–46, 1997.
5. P. Curzon, S. Tahar, and O. Ait-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 31–45, Australian National University, 1998.
6. P. Curzon. The Formal Verification of the Fairisle ATM Switching Element. Technical Report 329, Computer Laboratory, University of Cambridge, U.K., 1994.
7. L. A. Dennis, G. Collins, M. Norrish, R. Boulton, K. Slind, G. Robinson, M. Gordon, and T. Melham. The PROSPER Toolkit. In *Proceedings of the Sixth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. LNCS 1785, Springer Verlag, 2000.
8. M.J.C. Gordon. Combining Deductive Theorem Proving with Symbolic State Enumeration. 21 Years of Hardware Verification, December 1998. Royal Society Workshop to mark 21 years of BCS FACS.
9. M.J.C. Gordon and T.F. Melham. *Introduction to HOL:A Theorem Proving Environment for Higher-Order Logic.* Cambridge University Press, U.K., 1993.
10. J. Hurd. Integrating Gandalf and HOL. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics*, LNCS 1690, pages 311–321. Springer Verlag, 1999.
11. J.J. Joyce and C.J.H. Seger. Linking BDD-based Symbolic Evaluation to Interactive Theorem Proving. In *Proceedings of the 30th Design Automation Conference*, pages 469–474, Dallas, TX, June 1993.
12. R. Kumar, K. Schneider and T. Kropf. Structuring and Automating Hardware Proofs in a Higher-Order Theorem-Proving Environment. *Formal Methods in System Design*, 2:165–223, 1993.
13. I.M. Leslie and D.R. McAuley. Fairisle:An ATM Network for the Local Area. *ACM Communication Review*, 19(4):327–336, 1991.
14. S. Rajan, N. Shankar, and M.K. Srivas. An Integration of Model-checking with Automated Proof Checking. In Pierre Wolper, editor, *Computer Aided Verification*, Lecture Notes in Computer Science 939, pages 84–97. Springer Verlag, 1995.
15. K. Schneider and D.W. Hoffmann. A HOL Conversion for Translating Linear Time Temporal Logic to ω-Automata. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics*, LNCS 1690. Springer Verlag, 1999.
16. S. Tahar, X. Song, E. Cerny, Z. Zhou, M. Langevin and O. Ait-Mohamed. Modeling and Verification of the Fairisle ATM Switch Fabric using MDGs. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 18(7):956–972, 1999.
17. H. Xiong, P. Curzon, and S. Tahar. Importing MDG Results into HOL. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics*, LNCS 1690, 293–310. Springer Verlag, 1999.
18. M.H. Zobair and S. Tahar. On the Modeling and Verification of a Telecom System Block Using MDGs. Technical Report, Concordia University, Department of Electrical and Computer Engineering, December 2000.