

Scalable Automated Verification via Expert-System Guided Transformations

Hari Mony¹, Jason Baumgartner¹, Viresh Paruthi¹, Robert Kanzelman², and
Andreas Kuehlmann³

¹ IBM Systems Group, Austin, TX

² IBM Engineering & Technology Services, Rochester, MN

³ Cadence Berkeley Labs, Berkeley, CA

Abstract. *Transformation-based verification* has been proposed to synergistically leverage various transformations to successively simplify and decompose large problems to ones which may be formally discharged. While powerful, such systems require a fair amount of user sophistication and experimentation to yield greatest benefits – every verification problem is different, hence the most efficient transformation flow differs widely from problem to problem. Finding an efficient proof strategy not only enables exponential reductions in computational resources, it often makes the difference between obtaining a conclusive result or not. In this paper, we propose the use of an *expert system* to automate this proof strategy development process. We discuss the types of rules used by the expert system, and the type of feedback necessary between the algorithms and expert system, all oriented towards yielding a conclusive result with minimal resources. Experimental results are provided to demonstrate that such a system is able to automatically discover efficient proof strategies, even on large and complex problems with more than 100,000 state elements in their respective cones of influence. These results also demonstrate numerous types of algorithmic synergies that are critical to the automation of such complex proofs.

1 Introduction

Despite advances in formal verification technologies, there remains a large gap between the size of many industrial design components and the capacity of fully-automated formal tools. General exhaustive algorithms such as symbolic reachability analysis [1] are PSPACE-complete and limited to design slices with significantly fewer than one thousand state elements. Overapproximate proof techniques such as induction [2] are NP-complete and may be applied to significantly larger designs, though are often prone to inconclusive results in such cases. Consequently, even a piece of an industrial processor execution unit (much less an entire chip) is likely to be too large for a reliable application of automatic proof techniques.

Technologies such as bounded model checking (BMC) [3] and semi-formal verification [4, 5], which are generally NP-complete, have enabled leveraging the bug-finding power of formal algorithms to much larger designs. Though *incomplete*, hence generally unable to provide proofs of correctness, such applications have become prevalent throughout the industry due to their scalability and ability to efficiently flush out most

design flaws. Nevertheless, once such approaches exhaust their ability to find bugs, the end-user is often left debating how many resources to expend before giving up and hoping that the lack of falsification ability is as good as a proof.

The concept of *transformation-based verification* (TBV) [6] has been proposed to synergistically leverage various transformation algorithms to simplify and decompose large problems into sufficiently small problems that may be formally discharged. Though the complexity of a verification problem is not necessarily related to its size, the complexity class of the algorithms indicates an exponential worst-case relationship between these metrics, which is validated by practical experience. By resource-bounding any possibly costly BDD- or SAT-based analysis, it is possible to limit the complexity of most transformations used in a TBV system to polynomial while exploiting their ability to render exponential speedups to the overall verification flow [6, 7].

The strength of TBV is based upon the availability of a variety of different complementary transformations which are able to successively chip away at the verification problem until it can be handled by a terminal decision procedure. We have found that the power of TBV is often able to yield a proof for large problems which otherwise would be candidates only for falsification techniques. However, in cases, achieving such results requires a fair amount of user sophistication and trial-and-error – every verification problem is different, hence the most efficient transformation flow varies widely from problem to problem. Given a TBV system with a finite number of algorithms, each with a finite number of discretely-valued parameters, there are a countably infinite number of distinct proof strategies that could be attempted. Finding an efficient proof strategy not only entails exponential reductions in overall computational resources, it often makes the difference between obtaining a conclusive result or not.

In this paper, we propose the use of an *expert system* to automatically guide the flow of a transformation-based verification system. We discuss the rules used by the expert system to ensure that commonly-useful, low-resource strategies are explored first, then gradually more expensive strategies are attempted. We have found this approach useful for quickly yielding conclusive results for simpler problems, and efficiently obtaining more costly yet conclusive strategies for more difficult problems. We additionally discuss the type of feedback necessary between the TBV system and the expert system, needed to enable the expert system to effectively experiment with proof strategies. Lastly, we discuss the learning procedure used by the expert system to ensure that it leverages the feedback of previous experimentation in its quest for the best-tuned proof strategy for the problem at hand – ultimately seeking a conclusive result. Experimental results are provided to demonstrate that such a system is able to automatically yield proofs of correctness for large designs (with more than 100,000 state elements in the cone of influence) by maximally exploiting the synergy of the transformation and verification algorithms within the system against the problem under consideration.

Mechanizing the application of proof strategies is not a new concept; it is an essential component of most general-purpose theorem provers, e.g., HOL [8], PVS [9], and ACL2 [10]. However, the presented TBV approach is well-tuned for the verification of safety properties of hardware designs, incorporating numerous specialized transformations that are applicable to large systems. Finding a good scheduling and parameter setting for these transformations is non-trivial, though key to full automation.

2 Netlists: Syntax and Semantics

A *netlist* is a tuple $\langle\langle V, E \rangle, G, Z, T\rangle$ comprising a directed graph with vertices V and edges $E \subseteq V \times V$. Function $G : V \mapsto \text{types}$ represents a semantic mapping from vertices to gate *types*, including constants, primary inputs (i.e., nondeterministic bits), registers (denoted as the set R), and combinational gates with various functions. Function $Z : R \mapsto V$ is the initial value mapping $Z(v)$ of each register v ; note that this modeling allows symbolic initial values. The *semantics of a netlist* are defined in terms of semantic traces: 0, 1 valuations to gates over time which are consistent with G .

Our verification problem is represented entirely as a netlist, and consists of a set of *targets* $T \subseteq V$ correlating to a set of properties $AG(\neg t), \forall t \in T$. We say that target t is *hittable* if it evaluates to 1 along some trace, and that t is *unreachable* if no trace may hit t . We thus assume that the netlist is a composition of the *design under verification*, its *environment* (encoding *input assumptions*), and its *property automata*.¹

In our experiments, we map all designs onto a netlist representation containing only “constant zero” gates, two-input AND gates, inverters, and registers, using straight-forward logic synthesis techniques. Because inverters may be represented implicitly as edge attributes in the netlist representation [12], we assess the result of various transformation flows in terms only of register count, input count, and AND gate count.

3 Transformation-Based Verification

Transformation-based verification was proposed in [6] as a framework wherein one may synergistically leverage the power of various transformation algorithms to iteratively simplify and decompose complex problems until they become tractable for automated formal verification. All algorithms are encapsulated as *engines*, each interfacing via a common modular API. Each engine receives a verification problem represented as a netlist, then performs some processing on that problem. This processing could include an attempt to solve the problem (e.g., with a bounded model checking or reachability engine) or it could include an attempt to simplify or decompose the verification problem using a transformation (e.g. with a retiming or redundancy removal engine). In the latter case, unless the transformation itself solves the problem, it is generally desirable to pass the simplified problem to another engine to further process that problem. Note that the problem transmitted by an engine is generally not identical to the one received. Instead, as the problem flows from one engine to another, it is iteratively transformed into a simpler problem. As verification results are obtained on the simplified problem, those results propagate through the sequence of engines in the opposite order in which the netlist was transmitted, with each transformation engine undoing the effects of the transformations it performed to present its parent engine with results that are consistent with the netlist that parent transmitted. A particular instance of a TBV system is depicted in Figure 1.

The most useful verification and falsification engines are NP- to PSPACE-complete. In contrast, the applied transformations either require only polynomial resources or are

¹ Due to the ability to synthesize safety properties into automata [11], this invariant-checking model is rarely a practical limitation.

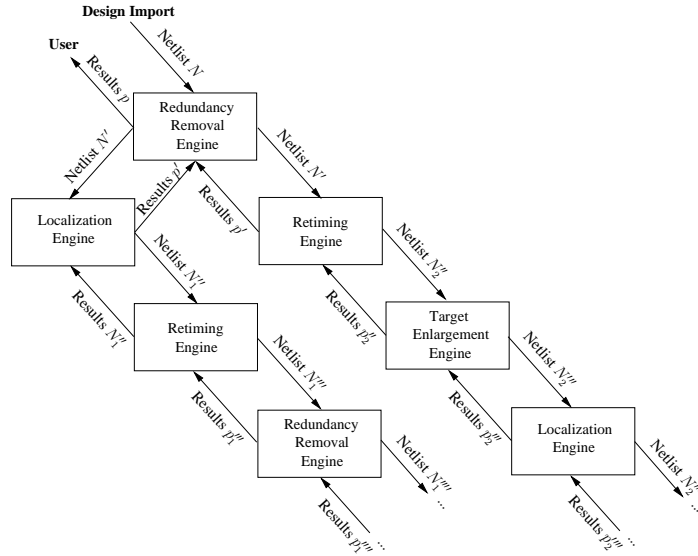


Fig. 1: Example flow of a transformation-based verification system

applied in a resource-constrained manner. They may ultimately reduce problem size by orders of magnitude compared to the initial size (even after a simple cone-of-influence reduction and constant propagations). We have found that the impact of such transformations on high-performance gigahertz-class designs is particularly pronounced, effectively automatically *undoing* many of the commonly employed high-performance microarchitecture and design techniques such as pipelining and addition of redundant logic to minimize propagation delays (e.g., replicating a lookup queue in two places in the circuit), which otherwise make the verification task of a given design component more difficult from one design generation to the next. A well-tuned transformation flow can therefore solve a problem with exponentially lesser computational resources than another flow, and even enable a conclusive result which otherwise would be infeasible.

3.1 Example Transformation Algorithms

In this section we introduce the transformation and verification engines used in our experiments. Recall that we measure netlist size in terms of register count, input count, and AND gate count. Each transformation implicitly performs a cone-of-influence reduction, hence may reduce all three metrics – though many explicitly attempt to reduce only one or two metrics, and may increase the others. Numerous options are available for each of these engines to bound resources and specify algorithmic parameters.

- **COM**: a redundancy removal engine, which attempts to merge functionally equivalent gates. This engine uses both on-the-fly compression techniques as the netlist is received (such as exploitation of an associative hashing routine to preclude the creation of redundant AND gates), as well as combinational semantic post-processing

techniques such as resource-bounded BDD- and SAT-based analysis (which otherwise are NP-complete) to identify functionally redundant gates [12]. This engine is guaranteed not to increase any of the three size metrics.

- **EQV**: another redundancy removal engine, similar to that proposed in [13]. This engine uses a variety of heuristics (such as symbolic simulation) to *guess* redundancy candidates, then uses an induction-based approach to prove and subsequently exploit that redundancy. Its reductions have the potential to far exceed those possible with **COM** and eliminate every redundant gate in the netlist; however, to achieve such exact reductions, the approach is PSPACE-complete, hence often lossy shortcuts must be accepted which trade reduction potential for run-time gains. This engine is guaranteed not to increase any of the three size metrics.
- **RET**: a min-area retiming engine [6, 14], which attempts to reduce the number of registers in the netlist by shifting them across combinational gates. This approach is guaranteed not to increase register count, but in calculation of retimed initial values via structural symbolic simulation, it may increase the other two metrics.
- **BIG**: a structural target-enlargement engine [15], which replaces a target by the characteristic function of the set of states which may hit that target within k time-steps, simplified with respect to the set of states which may hit that target in fewer than k time-steps. **BIG** is guaranteed not to increase register count nor input count, but may increase AND gate count.
- **CUT**: a range-preserving parametric-reencoding engine [7, 16], which replaces the fanin-side of a *cut* of the netlist graph with a trace-equivalent, yet simpler, piece of logic. **CUT** is guaranteed not to increase input count nor register count, but may increase AND gate count.
- **LOC**: a localization engine, which isolates a cut of the netlist local to the targets by replacing internal gates by inputs. This is similar to the processing of **CUT**, though in contrast, **LOC** does *not* preserve the range of the cut. This is the only transformation used in our experiments which is not both sound and complete – proofs of correctness on the localized design are valid for the unlocalized design, but counterexamples may be spurious (hence may need to be suppressed). To help guide the cut-selection process, the engine uses a light-weight SAT-based refinement scheme [17] to include only that logic which is deemed necessary. **LOC** is guaranteed not to increase register count nor AND gate count.
- **RCH**: an MLP-based symbolic reachability engine [18]. It is a general-purpose proof-capable engine, though PSPACE-complete.
- **SCH**: a semi-formal search engine [4, 5], which interleaves random simulation (to identify *deep*, interesting states), symbolic simulation (using either BDDs or a structural SAT solver [12]) to branch out from states explored during random simulation, and induction to attempt low-resource proofs of unreachability.

4 Tuning TBV Proof Strategies

Arriving at a well-tuned TBV engine flow with minimal effort is a nontrivial task for several reasons. The first is due to the number of possible proof strategies; given a system with k distinct engines, there exist k^i possible distinct engine sequences of length i .

Some engines are significantly more resource-intensive than others. It is therefore rarely an effective strategy to exhaustively attempt all possible engine flows of a certain depth. Instead, one will often wish to resort to a heuristic approach of partially exploring the *tree* of possible engine flows beginning with lower-cost, often-effective flows. One will then analyze their effectiveness upon the corresponding problem (e.g., their achieved reduction) to decide what to attempt next – whether to pursue appending further transformations onto the end of some of those already-explored flows, or to try some new, possibly more expensive flows in the quest for obtaining a conclusive result. Viewed another way, the user prioritizes among each node of the explored tree based upon the size of the problem at that node and the suspected reduction potential beyond that node (e.g., if a deeper transformation flow is attempted beyond that node), and systematically chooses promising nodes from which to further experiment.

The second difficulty is due to the irreversibility of certain transformations. Several verification-oriented transformations – particularly approximate ones such as localization – alter the netlist in a manner which may not be readily reversible.² Thus, applying transformations *A* then *B* may yield a different netlist than applying *B* then *A*, and performing a certain transformation may destroy the ability to subsequently solve the problem without backtracking out of that transformation. This precludes a simple depth-first search from being an effective proof strategy; some degree of branching and bounding must be performed. To compound this problem, recall that there are three netlist size metrics that we consider; many transformations reduce one or two of these, and may substantially increase the others. Different algorithms are more sensitive to some of these metrics than others – e.g., symbolic reachability analysis is highly sensitive to register count whereas structural SAT solvers are more sensitive to AND gate count. This overall makes it difficult to rate the effectiveness of a given transformation.

Given the amount of experimentation necessary to solve complex verification problems, there exists a strong motivation to attempt to automate this overall process. This automation may save considerable manual effort for expert users, furthermore enabling a much faster time to a conclusive result due to the ability to automatically explore proof strategies in parallel. Additionally, it enables even casual users (for example, a logic designer who is not versed in formal methods) to obtain results that otherwise would not be obtainable, at least until the problem is transferred to an expert user of the tool. Finally, an automated process may well be able to effectively *learn* strategies and algorithmic synergies that may otherwise go unexplored even by an expert user.

5 Automating TBV Proof Strategies via an Expert System

In this section, we describe how to fully automate the process of obtaining efficient proof strategies with a TBV system. The overall architecture of this automated system is depicted in Figure 2. At the center of this system is the TBV core itself. Problems are imported into the TBV system, and the results are reported, via the *User Interface*. The verification process itself is controlled via the *Proof Strategy Interface* which includes the *Engine Control Interface* through which the engine selection process is performed,

² The process of *reversing a transformation* is often emulated by removing the corresponding engine from a given transformation flow.

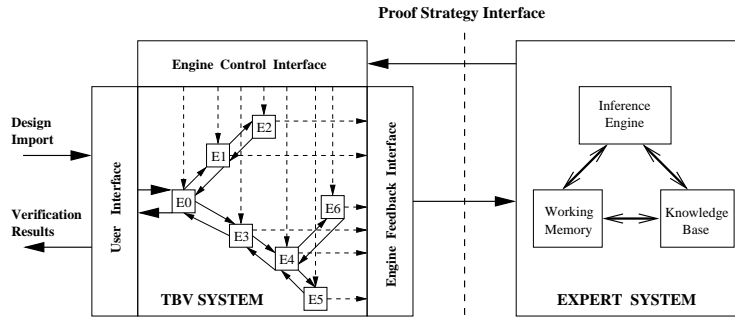


Fig. 2: Integration of an expert system with a TBV system

and the *Engine Feedback Interface* through which feedback is reported to indicate how well the various engines are performing upon the active problem.

An example run of the TBV system is depicted in Figure 2. The problem is first processed using the proof strategy of engine E0, then E1, then E2, each selected via the *Engine Control Interface*. Next, assume that, based upon the *Feedback* received from these engines, the controller of the verification process wishes to attempt a different proof strategy. The controller will thus preclude attempting to further transform the problem after E2, and instead instruct the TBV system to pass the transformed problem from E0 to a new engine flow beginning with E3.³

The control of the TBV process is traditionally carried out by running pre-packaged strategies, which is limited in that a conclusive strategy for a hard problem may not be previously known. Alternatively, it may be put under the control of a user who will wish to manually tune the flow based upon *Feedback* provided by the system, using the heuristic experimentation process described in Section 4. To achieve full automation of the verification process while retaining the ability to experiment to yield conclusive results on difficult problems, we propose to eliminate the need for user interaction by attaching an *expert system* to the *Proof Strategy Interface*.

5.1 Expert Systems

Feigenbaum [19] defined an expert system as “an intelligent computer program that uses knowledge and inference procedures to solve problems that are difficult enough to require significant human expertise for their solution.” Expert systems are often used to solve complex problems with ill-defined domains for which no algorithmic solution is known or which belong to an intractable class of problems (such as NP- or PSPACE-

³ It is an implementation-specific detail as to whether the TBV system will retain the branch containing engines E1 and E2 in scope. Retaining engines outside of the active proof path increases overall resource consumption, though facilitates subsequent experimentation along those prior branches without needing to re-run those flows. Note also that the exploration of distinct branches may be performed by multiple machines operating in parallel, possibly transferring *snapshots* of the transformed netlist.

complete). Countless distinct types of expert system applications have been proposed over the past decades.

The three main architectural components of an expert system are the following.

- The *knowledge base*, which contains the domain-specific knowledge used to solve problems. This knowledge can be elicited from human experts, or it can be learned by the system itself. Knowledge is often represented in the form of *rules* which govern the solution strategies employed by the expert system.
- The *working memory*, which refers to task-specific data for the problem under consideration. The working memory is the data that is read and written to as the expert system attempts to solve the problem.
- The *inference engine*, which contains the algorithms used to leverage the rules in the knowledge base in order to solve problems. Once the knowledge base is built and the task-specific data is read into the working memory, the inference engine will start evaluating rules to attempt to solve the problem.

5.2 Expert System Implementation

In this section we describe how to implement an expert system for deriving effective proof strategies for a TBV system. Figure 3 depicts the high-level experimentation algorithm. As the expert system experiments with the problem and partially explores the tree of possible proof strategies, it records data learned from that experimentation in a database called the *Tree of Knowledge* during step 3. As discussed in Section 4, there are two primary decisions involved in the experimentation process: choosing a promising node in an engine flow from which to further experiment (decided during step 1), and selecting the next engine to append onto that chosen node (decided during step 2).

The Tree of Knowledge. The *Tree of Knowledge* is a database of information learned during prior experimentation on the active problem, which may generally comprise results obtained in a parallel-processing environment. Each node in the tree corresponds to the run of a particular engine instance (including its options) and the *Feedback* received from that run. The recorded *Feedback* information may include the following.

1. The transformed netlist size, and information about any solved targets.
2. The resources consumed by that engine run.
3. Dynamically-obtained information on how an engine’s options could be improved. This feedback includes two aspects.
 - (a) How to initialize the system to a given node with lesser resources. For example, SAT-solvers use various heuristics in their processing; the best heuristic for a given problem may enable dramatic speedups vs. another heuristic. If a SAT-based engine determines during its run that a particular heuristic worked best, it may report that information to be recorded in the *Tree of Knowledge* so that a future run can initialize to this node more quickly.
 - (b) *Hints* to yield a superior flow. For example, if a redundancy removal engine sees that it precluded some analysis due to resource limitations, likely hurting its reduction potential, it would provide feedback so that a future run could increase the corresponding type of resources and yield superior reductions.

```

while (¬solved) {
  1. Choose a node from the Tree of Knowledge from which to perform further experimentation;
  initialize the TBV system to that node (call this engine  $E_i$ ).
  2. Choose an engine  $E_{i+1}$  to append to the initialized node; instruct engine  $E_i$  to pass its
  transformed problem to  $E_{i+1}$ , then run engine  $E_{i+1}$ .
  3. Extract feedback from the run of  $E_{i+1}$ ; update the Tree of Knowledge with this data.
}

```

Fig. 3: High-level expert system algorithm

Choosing a node for further experimentation. During step 1 of the algorithm of Figure 3, the expert system chooses a node from the *Tree of Knowledge* from which to perform further experimentation, then initializes the TBV system into the corresponding state. To make this decision, the expert system assigns a *priority* to each node in the *Tree of Knowledge* using a set of rules; it then performs a weighted random selection among the prioritized nodes. Several classes of rules are used to assign priorities.

- **Prefer smaller netlists.** An important set of rules is based upon the size of the netlist at corresponding nodes in the tree – the smaller the problem, the closer to a conclusive result the system tends to be, though this trend is certainly not guaranteed. However, recall that there are three distinct size metrics considered. The size-related priority ascribed to a node is thus varied during the experimentation, from strategies using equal weights among all the metrics, to strategies using brute-force to minimize the number of registers (without regard to other metrics) in the hope of yielding an inductive target or one which is amenable to reachability analysis, to more experimental strategies which prioritize towards nodes which managed to significantly reduce *any* of the metrics in the hope that a subsequent flow may compensate for increases in the other metrics and ultimately yield a proof.
- **Prioritize away from well-explored branches.** Based upon the previous set of rules, the system will tend to perform near-exhaustive exploration of the first explored branch consisting of strictly decreasing size metrics. While this is indeed an advantageous strategy to begin with, it may preclude necessary experimentation along a completely different flow to yield a complex proof. Therefore, a class of rules is needed to prioritize away from nodes in the tree that have been more thoroughly explored – i.e., deeper exploration is no longer yielding significant reductions despite having experimented with most advantageous engine types.
- **Exploit low-cost re-initializations.** Note that there is a computational cost associated with initializing the TBV system to a given node (that is not already in scope), namely that of re-running the desired transformation flow up to the desired node. It is therefore often advantageous to have a set of rules prioritizing towards a fair amount of experimentation from or near the active nodes before re-initialization into completely different branches.
- **Identify the causes of unsuccessful branches.** In cases, the reason that a proof cannot be obtained along a branch may be attributed to a transformation which occurred much earlier in the flow. For example, if an instance of **LOC** subsequently renders a spurious counterexample, it is of no utility to perform further experimen-

tation under that instance. As another example, it may be the case that the problem at a branch is suffering from far too many inputs to complete a reachability computation and no available algorithms are able to compensate for that metric, and that an earlier instance of **RET** was the cause of a significant increase in input count. Though the **Prioritize away from well-explored branches** class of rules will ultimately bring us out of heavy exploration of such branches, another class of rules is useful to attempt to further leverage information about the *cause* of unsuccessful branches, and backtrack sufficiently far to circumvent that causal engine, instead of continuing to branch to other points under that causal engine.

Choosing an engine to append to the initialized node. During step 2 of the algorithm of Figure 3, the expert system must decide what engine type to append to the initialized node. Because any engine type may be run at any given time, rules are again deployed to determine priorities for these possibilities; the expert system then performs a weighted random selection among the prioritized possibilities. The following classes of rules are useful to prioritize among the possible choices.

- **Begin with low-cost flows.** The primary objective is to obtain verification results as quickly as possible, so the initial priorities are set to attempt commonly-effective, low-cost flows first. Whether falsifying or proving, reduction algorithms can yield dramatic improvements in runtimes and enable conclusive results that otherwise may be missed given available resources, so the priorities of low-cost reductions such as **COM** and **RET** are initially high. Additionally, the priorities of low-cost falsification and proof engines such as **SCH** are initially high.
- **Gradually attempt more expensive flows.** As more and more experimentation is performed, the expert system acknowledges that the problem is increasingly more difficult, hence gradually increases the priority of its heavier-weight reduction algorithms such as **EQV**. Additionally, as more and more attempts at falsification render inconclusive results, the priority of proof-capable flows including **LOC** and **RCH** are increased and the priority of falsification engines such as **SCH** are decreased.
- **Exploit known algorithmic synergies.** Known synergies between transformations should be exploited by a set of rules. For example, running a redundancy removal engine after retiming often helps reduce the potential increase in combinational logic caused by that engine, hence priorities should be adjusted accordingly.
- **Explore the potential of unknown algorithmic synergies.** Because every problem is different, it is generally impossible to predict the possible algorithmic synergies that will be key to solving that problem, hence the system should have a set of rules to attempt to prioritize towards such experimentation. It is rarely useful to re-run the same engine type back-to-back, without any intermediate transformations. However, it is often the case that repeated calls of a given transformation engine, interspersed with other transformations, is an effective strategy whereby one transformation synergistically unlocks further reduction potential for another. A generic way to encode this trend is to analyze the extent to which the size metrics changed since the last run of a given engine type, and update the priority for instantiating that engine type accordingly. For example, assume that the current engine flow is **RET**, **COM**, **CUT** where **CUT** performed slight reductions, whereas

COM performed significant reductions. The priority of appending **RET** to this flow is increased since the netlist changed significantly in size since **RET** was last run, whereas the priority of appending **COM** is decreased due to the lack of change in netlist size since that engine last ran. Note that such experimentation may result in the eventual learning of commonly useful known synergies – this has the potential to enable the overall system to grow in effectiveness as it is deployed upon more and more problems, especially as new engine types are added to the system.

- **Leverage the Tree of Knowledge.** The *Tree of Knowledge* contains information about prior experimentation. Because the concept of initializing the system into a previously-explored node is covered by step 1, re-running an already attempted child engine of the initialized node is lowered in priority in this step, particularly if that child did not yield a beneficial reduction. Also, the *hints* recorded from prior runs may be used to attempt to obtain an improved proof strategy.

6 Experimental Results

In this section we provide the experimental results of the integration of an expert system with a TBV system. All experiments were run on an IBM RS/6000 Model 43P-S85 (850MHz), using a single processor and 4GB main memory. In addition to the engines discussed in Section 3.1, we utilized a phase abstraction engine [20] on all IBM designs prior to importing them into the TBV system.

We had intended to provide a large set of results showing the run-time difference of various proof strategies; however, the majority were easily discharged by a straightforward proof strategy. For example, we ran the 42 designs of the ISCAS89 test suite using each primary output as a target.⁴ Except for one target of S635, the overall system was able to discharge all 1615 resulting targets using the proof strategy **COM, RET, COM, SCH, BIG, RCH** in less than 35 cumulative minutes of runtime.⁵ We therefore consider these to be easy problems; rather than describing these experiments in more detail, we thus turn our attention to significantly more difficult problems.

Table 1 comprises some of the most difficult verification problems we have encountered among IBM designs. The first column is a label column, indicating the name of the corresponding design and the metric being tracked in the corresponding row. The second reflects the size of the original, unreduced verification problem. The successive columns indicate the size of the problem *after* the corresponding transformation engine (indicated in the row labeled with the design name) was run. The total run-time, memory consumption, and result of the corresponding proof strategy is also provided.

MFC is a memory flow controller. ERAT is an effective-to-real address translation unit. IOC is an I/O Controller. RING comprises starvation and prioritization correctness properties for network arbitration logic. SQM is an InfiniBand store queue manager. MMU and SMM are different memory management units; we report results for several different types of properties for these. All of these had undergone many months of simulation-based analysis prior to importing into the TBV framework, and weeks of

⁴ Though these may not be meaningful properties to check of these designs, none are otherwise available for them; these are additionally easily-reproducible experiments.

⁵ The target of S635 requires 2^{32} inexpensive image computations to hit.

straight-forward transformations, semi-formal falsification, and proof analysis had been performed within the TBV system without conclusive results prior to deploying the expert system. SMM and SQM were the most difficult of these examples, requiring nearly four days of experimentation by the expert system (running one thread). RING was the second most difficult, requiring nearly two days. The others were solved within one day of experimentation. We noted the following trends in our experiments.

- **LOC** performs an overapproximate transformation; the choice of too small of a localized cone may render a spurious counterexample. More generally, localization locks us into a netlist where redundancy removal techniques may be weakened: e.g., gates which are constant in the original netlist may not be constant in the localized netlist. It is thus important to choose a large enough cone not only to prevent spurious counterexamples, but also to prevent weakening other redundancy removal transformations which may ultimately be needed to yield a conclusive result. It therefore came as a surprise to us that the expert system found, after some experimentation, that some of our hardest problems were efficiently solvable when performing localization after only basic reductions. For these cases, early localization was a successful strategy since it quickly identified a sufficiently large cone that remained unreachable, yielding a significantly smaller netlist against which we could successfully leverage more costly reductions. Note that nested localizations intermixed with other transformations often yields increasing reductions.⁶
- **RET** often substantially reduces register count, though may significantly increase combinational logic due to the symbolic simulation necessary to compute retimed initial values. While **COM**, **CUT**, and **EQV** are useful in offsetting this increase, as one adds more **RET** calls, more and more of the registers begin to attain symbolic initial values as reachability data is effectively locked into their initial values. Some of these runs show that the expert system found it effective to resort to a more costly **EQV** coupled with fewer retiming runs accordingly.
- Redundancy removal is almost always a useful transformation. **EQV** may yield dramatic reductions, though tends to be the most expensive transformation discussed if used with sufficiently high resources to yield near-optimal reductions.
- A general characteristic of TBV is that one reduction often enables another, in turn allowing the same transformation to yield increasing reductions when interspersed with other transformations. For example, redundancy removal, parametric re-encoding, and localization are able to break connections which constitute retiming traps, enabling multiple retiming instances to yield increasing reductions as noted in [6]. Redundancy removal often enables parametric reduction that otherwise could not be obtained [7]. Retiming may enable a gate that acts as a constant only after the first several time-steps of execution to be merged as a constant in a sound and complete fashion [7], enabling further redundancy removal.

For MFC, the flow of **RET** before **LOC** was critical; without the simplification enabled by retiming, the localized cones became hopelessly large. ERAT and IOC com-

⁶ A similar observation on the utility of nested localizations was noted in [21], applied in an approach which extracts an *unsatisfiable core* from a BMC SAT instance. Our application in a TBV domain yields greater flexibility in its ability to leverage various transformations between the localization instances.

prised a large degree of redundant logic. Both **EQV** and **RET** are required to solve these targets, though the former is much more expensive than the latter – it was found that the combinational gate increase by **RET** caused the best strategy to place that engine after **EQV**. RING.P required multiple retiming passes to render an inductive target. RING.S was an interesting case; our last **EQV** call was a thorough and costly one, requiring nearly five hours of run time. Once reduced, we found a valid counterexample of depth 192 on the resulting design in less than one minute. Without this reduction, we found that we could not complete the localization refinement of the failing time-step within a period of 36 hours, without which the localized cone yielded spurious counterexamples. This illustrates the power of TBV not only to enable difficult and large proofs, but to yield exponential speedups to the bug-finding process. SQM required aggressive redundancy removal to enable a highly-effective retiming and localization step, which in turn enabled the property to be trivialized during a subsequent aggressive redundancy removal step. SMM was a difficult problem for which we have not found a shorter proof-capable strategy, though its overall run-time was quite reasonable. The winning strategy was to iteratively leverage all of the discussed transformations to chip away at the netlist size, even requiring what seemed to be rather poor engine choices (e.g., note that **RET** was instantiated numerous times yielding a reduction of only one register, at the cost of tripling input count). Additionally, that strategy chose many low-resource calls to **CUT** which yielded only modest reductions; larger-resource calls yielded much greater input reductions, but at the cost of much greater AND gate increases which slowed the overall flow. Ultimately, the synergy between the algorithms rendered a sufficiently small netlist to enable a proof by induction or reachability.

7 Acknowledgements

The authors would like to thank Geert Janssen, Mark Williams, and Jessie Xu for their contributions to numerous aspects of the TBV system, as well as for insights into the integration of the expert system into this framework.

8 Conclusion

In this paper we propose the use of an *expert system* to automate the experimentation necessary to obtain an efficient proof strategy for a *transformation-based verification* system [6]. We discuss details of how to integrate an expert system with a TBV system, and of how to customize the expert system to enable it to yield conclusive verification results with minimal resources. Our experiments demonstrate the utility of such an overall system in automatically yielding complex proofs for large industrial designs with over 100,000 registers in their cone of influence, even after simple redundancy removal. This integration eliminates the need for manual effort or formal expertise to yield conclusive results on such difficult problems. Additionally, this integration enables the exploitation of parallel processing to automatically *learn* algorithmic synergies critical to the solution of such complex problems which otherwise may go undiscovered.

References

1. O. Coudert, C. Berthet, and J. C. Madre, "Verification of synchronous sequential machines based on symbolic execution," in *Int'l Workshop on Automatic Verification Methods for Finite State Systems*, June 1989.
2. M. Sheeran, S. Singh, and G. Stalmarck, "Checking safety properties using induction and a SAT-solver," in *Formal Methods in Computer-Aided Design*, Nov. 2000.
3. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Tools and Algorithms for Construction and Analysis of Systems*, March 1999.
4. M. Ganai, A. Aziz, and A. Kuehlmann, "Enhancing simulation with BDDs and ATPG," in *Design Automation Conference*, June 1999.
5. P.-H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long, "Smart simulation using collaborative formal and simulation engines," in *Int'l Conference on Computer-Aided Design*, Nov. 2000.
6. A. Kuehlmann and J. Baumgartner, "Transformation-based verification using generalized retiming," in *Computer-Aided Verification*, July 2001.
7. J. Baumgartner, *Automatic Structural Abstraction Techniques for Enhanced Verification*. PhD thesis, University of Texas, Dec. 2002.
8. M. Gordon, "Mechanizing programming logics in higher order logic," in *Current Trends in Hardware Verification and Automated Theorem Proving*, Springer-Verlag, 1989.
9. M. Srivas, H. Rueß, and D. Cyrluk, "Hardware verification using PVS," in *Formal Hardware Verification: Methods and Systems in Comparison*, Springer-Verlag, 1997.
10. M. Kaufmann, P. Manolios, and J. S. Moore, *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.
11. E. A. Emerson, "Temporal and modal logic," *Handbook of Theoretical Computer Science*, vol. B, 1990.
12. A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *IEEE Transactions on Computer-Aided Design*, vol. 21, no. 12, 2002.
13. P. Bjesse and K. Claessen, "SAT-based verification without state space traversal," in *Formal Methods in Computer-Aided Design*, November 2000.
14. J. Baumgartner and A. Kuehlmann, "Min-area retiming on flexible circuit structures," in *Int'l Conference on Computer-Aided Design*, Nov. 2001.
15. J. Baumgartner, A. Kuehlmann, and J. Abraham, "Property checking via structural analysis," in *Computer-Aided Verification*, July 2002.
16. I.-H. Moon, H. H. Kwak, J. Kukula, T. Shiple, and C. Pixley, "Simplifying circuits for formal verification using parametric representation," in *Formal Methods in Computer-Aided Design*, Nov. 2002.
17. P. Chauhan, E. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang, "Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis," in *Formal Methods in Computer-Aided Design*, November 2002.
18. I.-H. Moon, G. D. Hachtel, and F. Somenzi, "Border-block triangular form and conjunction schedule in image computation," in *Formal Methods in Computer-Aided Design*, Nov. 2000.
19. E. A. Feigenbaum, "Themes and case studies of knowledge engineering," in *Expert Systems in the Micro-Electronic Age*, 1979.
20. J. Baumgartner, T. Heyman, V. Singhal, and A. Aziz, "An abstraction algorithm for the verification of level-sensitive latch-based netlists," *Formal Methods in System Design*, no. 23, 2003.
21. A. Gupta, M. Ganai, Z. Yang, and P. Ashar, "Iterative abstraction using SAT-based BMC with proof analysis," in *Int'l Conference on Computer-Aided Design*, Nov. 2003.