SAT-based Induction for Temporal Safety Properties

Roy Armoni [a,1] Limor Fix [a,1] Ranan Fraer [a,1]
Scott Huddleston [b,1] Nir Piterman [a,1] Moshe Y. Vardi [c,2,3]

[a] *Design Technology – Intel, Haifa, Israel*

[b] *Desktop Product Group – Intel, Hilsboro, Oregon*

[c] *Dept. of Computer Science, Rice University*

Abstract

The work presented in this paper addresses the challenge of fully verifying complex temporal properties on large RTL designs. Windowed induction has been proposed by Sheeran, Singh, and Stalmarck as a technique augmenting Bounded Model Checking for unbounded verification of safety properties. While induction proved to be quite effective for combinational properties, the case of temporal properties was not handled by previously known methods. We introduce *explicit induction*, a new induction scheme targeted to temporal properties, and to interactive development of inductive proofs. The innovative idea in explicit induction is to make the induction scheme an explicit part of the specification, where it can be easily controlled, using a highly expressive language like ForSpec. We show how explicit induction was implemented with minor modifications in the ForSpec compiler and in Thunder, a bounded model checker. Finally, we describe how explicit induction was used for verifying large control circuits with extensive feedback in the Pentium™4 processor. The circuits verified by explicit induction are orders of magnitude larger than those verifiable by traditional model checking approaches.

Key words: SAT, induction, windowed induction, safety

1 Introduction

The general aim of formal verification is to provide compelling evidence of the correctness of a system in the form of a mathematically precise argument

[1] Email: firstname.lastname@intel.com

[2] Email: vardi@cs.rice.edu

[3] Supported in part by NSF grants CCR-9988322, CCR-0124077, CCR-0311326, IIS-9908435, IIS-9978135, EIA-0086264, and ANI-0216467 by BSF grant 9800096, and by a grant from the Intel Corporation.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

showing that the system (implementation) satisfies a collection of required properties (specification). In model checking, we verify the correctness of finite-state systems with respect to a desired behavior by checking whether a labeled state-transition graph that models the system satisfies a temporal logic formula that specifies this behavior [6]. Model-checking has two major advantages, namely, it is fully automatic, and in the case of failure produces a counterexample (an erroneous execution of the system).

The introduction of symbolic model checking based on BDDs [4,10] has increased the capacity of model checking and made it a standard in hardware industry [1]. BDDs are a canonic representation of Boolean functions and are used to represent sets of states and transitions of the model. BDD-based model checkers compute the set of reachable states (or sometimes analyze cycles) to ensure that there are no disallowed behaviors. In spite of the increased capacity, it soon became apparent that *state explosion* is still a problem. A major breakthrough has been the introduction of bounded model checkers [2]. Bounded model checking is based on the representation of computation paths falsifying the specification in the form of a Boolean satisfiability problem. The usage of bounded model checking increased the size of models handled by model checkers; however, at a price. We no longer get a fully certified answer to the verification problem but rather assurance that there are no counterexamples of a given length. This observation makes bounded model checking especially adequate for bug hunting. Still there is a limit on the size of bounds handled by bounded model checkers, leaving us with lack of complete assurance in the correctness of the design under verification.

In practice, most specifications are *safety* properties. A property is called *safety* if we can deduce that it is false by examining a finite computation path. Combinational properties of the form `ALWAYS` p, also called *invariants*, are a particular case of safety properties that have two distinctive characteristics:

- Every safety property can be reduced to an invariant by a compilation process described below [8].
- Some invariants can be fully proved with a powerful technique, called *induction* [13].

Besides being a complete proof technique, induction has a better capacity than bounded model checking, as it has to unroll the model only to a small depth. The following paragraph explains how the induction works.

1.1 Induction for Invariants

Traditional mathematical induction can be used to prove that a property $P(n)$ holds for all nonnegative integers n. An induction proof consists of proving the following two subgoals:

- Prove that $P(0)$ is true.
- Prove that forall k, $P(k)$ implies $P(k+1)$.

In formal verification, induction has been used to prove an invariant P in a transition system by showing that P holds in the initial states of the system and that P is maintained by the transition relation of the system [9]. In many cases P is not inductive by itself, and one has to find a strengthening of P that is inductive.

More formally, let $M = (S, S_0, T)$ be a transition system, where S is a set of states, $S_0 \subseteq S$ is a set of initial states and $T \subseteq S \times S$ is a transition relation. For simplicity of presentation, we relate to sets of states as predicates, e.g., by using the characteristic function of the set. The classical induction methodology for proving P is based on manually finding a property Q (the *induction hypothesis*) such that $Q \Rightarrow P$ and proving the following two subgoals:

- The initial states of M satisfy Q: forall states x_0, we have $S_0(x_0) \Rightarrow Q(x_0)$
- Q is maintained by the transition relation: forall states x_0 and x_1 we have $Q(x_0) \wedge T(x_0, x_1) \Rightarrow Q(x_1)$

This classical method is known to be theoretically sound and complete, where theoretical completeness is demonstrated by having the property Q describe the set of reachable states of M. Note that induction hypotheses are typically much simpler than a full reachable state description. When it succeeds, induction is able to handle larger models than bounded model checking, since the induction step has to consider only paths of length 1, whereas bounded model checking needs to check sufficiently long paths to get a reasonable confidence [3].

1.2 Windowed Induction

In many cases, constructing an inductive invariant for simple induction is not feasible. Windowed induction is a modified induction technique, which can considerably simplify finding inductive invariants for proofs on hardware models. Mathematically, windowed induction with window size $N \geq 0$ consists of the following two steps:

- Prove that for $0 \leq k \leq N$, $P(k)$ is true.
- Prove that forall k, $(P(k) \wedge \ldots \wedge P(k+N)) \Rightarrow P(k+N+1)$.

Windowed induction proofs in a hardware system are realized as follows [13]. To prove that P is an invariant of system M, we do the following:

(i) Manually find a strengthening Q of P for which Q implies P. Typically we choose Q to be $P \wedge \langle \text{something} \rangle$.

(ii) Find an N for which the following two proofs are achievable:

   (a) Base: Q holds in all paths of length N starting from an initial state:

$$S_0(x_0) \wedge T(x_0, x_1) \wedge \ldots T(x_{N-1}, x_N) \Rightarrow Q(x_0) \wedge Q(x_1) \wedge \ldots \wedge Q(x_n)$$

   (b) Step: For an arbitrary path of length $N+1$, if Q holds in the first

$N+1$ states, then it holds in state $N+2$ too

$$T(x_0, x_1) \wedge \ldots \wedge T(x_N, x_{N+1}) \wedge Q(x_0) \wedge \ldots \wedge Q(x_N) \Rightarrow Q(x_{N+1})$$

This method is also known to be sound and complete. Even without strengthening P, if we restrict the induction step only to loopfree paths, completeness can be proved by choosing N to be the recurrence diameter of the transition system M, i.e., the maximum length of a loopfree path in M. The advantage of windowed induction over classical induction is that it provides the user with two ways of strengthening the induction hypothesis: strengthening the invariant Q or lengthening the window N. (For simplicity, we do no mention the loopfreeness condition in the rest of our discussion, but it is implemented in our tool.)

Windowed induction is used in [13], and is considered more abstractly in [5]. The formal verification environment in Intel offers this induction scheme as an automatic mode in the SAT-model checker Thunder. The bound N is iteratively increased until either the proof succeeds or a given limit is reached. Windowed induction and standard induction have the same theoretical capabilities, but windowed induction often permits much simpler induction hypotheses. In many cases the size of windows is relatively small. As with simple induction, we get the best of both worlds: we get a correctness proof and we get the ability to handle very large models.

Intuitively, pipelines in sequential hardware circuits are why windowed induction proofs can use simpler induction hypotheses than non-windowed induction proofs of the same property. When a property P of interest depends on a pipeline of depth d, a windowed induction with window size d can sometimes prove P inductively without strengthening P. But a comparable standard induction proof in this case generally has to strengthen P to express many internal invariants on the pipeline in order for the proof to succeed.

The techniques in [13] have been used successfully at Intel, as they proved to be quite effective for verifying combinational properties. One benefit of this approach is that it automates much of the induction mechanism, including automatically searching for a working induction window size.

1.3 Implicit Induction for temporal properties

When a property P is not combinational, but rather a complex temporal specification such as a typical formula that is written using the specification language ForSpec, it may not be obvious how to prove P by induction. As mentioned, these assertions are often safety properties. The ForSpec compiler synthesizes P into an automaton A_P and an invariant Z_P [14,8] such that for every transition system M:

$$M \models P \text{ iff } M_P \models \texttt{ALWAYS } Z_P$$

Where $M_P = M \parallel A_P$ denotes the synchronous composition of M and A_P. The above observation about the behavior of the ForSpec compiler, immediately suggests that for a safety property P, the classical induction methodology may be used to prove $M \models P$ by proving that $M_P \models$ `ALWAYS` Z_P.

As an example, consider the ForSpec property `ALWAYS` $\neg(f, f)$, which forbids two consecutive occurrences of f, where f is a combinational property. When compiling its negation, the property `EVENTUALLY` (f, f), we can get, for example, the three-state automaton in Figure 1. This automaton can cycle in its initial state s_0 or move nondeterministically to s_1 upon receiving the input f. A second occurrence of f is needed to move from s_1 to the accepting state s_2, while an occurrence of $\neg f$ brings us back to s_0. The output Z marks the accepting state of this automaton, such that $Z = 0$ if and only if the original property `ALWAYS` $\neg(f, f)$ fails. If we manage to prove `ALWAYS` Z by induction, then we have also a complete proof for `ALWAYS` $\neg(f, f)$.

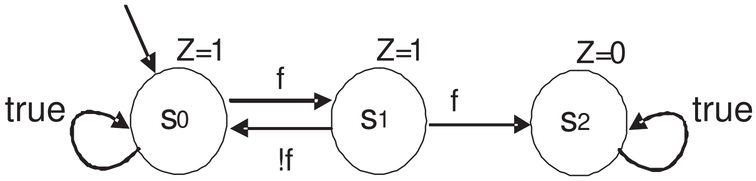

Fig. 1. Accepting automaton for `EVENTUALLY` (f, f) - the negation of `ALWAYS` $\neg(f, f)$

This approach is implemented at Intel on top of Thunder, a bounded model checker [7]. When invoking the induction algorithm, the tool searches for a large enough window size N for which both requirements (ii-a) and (ii-b) hold. Completeness guarantees that such an N exist, even without strengthening the invariant. In practice, the induction succeeds with a reasonably small N for many combinational invariants. For temporal properties expressed in ForSpec, however, the tool typically fails to complete the proof. The problem is both algorithmic and methodological, as we now explain.

On the algorithmic side, the approach suffers from a serious capacity issue. Consider again the property $P =$ `ALWAYS` $\neg(f, f)$ and the automaton in Figure 1. Assume that this property can be proven by classical induction, i.e., the initial states of M satisfy $\neg(f, f)$ and $\neg(f, f)$ is maintained by the transition relation. Thus, this property passes with a window $N = 1$. However, `ALWAYS` Z cannot be proven by induction with a small window N. For every small N, there is a path of length $N + 1$ failing the induction step (the path looping $N - 1$ times in s_0 before taking a transition to s_1 and then to s_2). Although this path contains loops in the automaton A_P alone, it can well be a loopfree path in the product $M_P = M \parallel A_P$. In the worst case, the minimal window N for which the induction succeeds is the recurrence diameter of the system M (the minimal length of loopfree paths in M), which usually exceeds the capacity of the tool. (It might seem that the problem is caused by the specific automaton used in the example, but this is not the case.)

One way to reduce the window size is to strengthen the invariance Z_P

manually. This, however, requires expressing an invariant over the augmented design M_P. While the user can be expected to have an understanding of the internal details of the system M, the user cannot be expected to have an understanding of the internal details of the automaton A_P, which is the output of the ForSpec compiler. The user can even less be expected to understand the interaction of M with A_P. Thus, requiring the user to generate invariances of M_P is not realistic. This suggests that implicit induction, even with manual intervention, cannot be effectively used for verifying temporal ForSpec properties.

1.4 Explicit Induction for Temporal Properties

This limitation is in fact what motivated our work. We want to perform the inductive reasoning directly on the original temporal formula, rather than on the results of its compilation. To this end, we encode the induction scheme *explicitly* as part of the specification. A failure to prove the induction produces a meaningful counterexample, which reflects the real reason of the failure and is not due to compilation artifacts anymore. Understanding the induction failure is a crucial hint for strengthening the inductive property.

In our opinion, this is the only effective way to allow the manual guidance of the user in the iterative process of finding the right induction hypotheses. In this respect, we take the approach of [13] one step further, and to our knowledge this is the first attempt to perform induction directly on temporal properties.

To be more precise, consider again a transition system $M = (S, S_0, T)$ and denote by $uninit(M)$ the non-initialized model (S, S, T) in which every state in S is an initial state.

Consider a ForSpec formula P. We say that P is *bounded* if there exists k such that for every path π, the truth value of P on π can be determined by considering the prefix of π of length k. For example, the truth value of $a \wedge$ `NEXT`$[5]$ b (that is, a holds now and b holds after 5 time units) can always be determined by considering a prefix of length 6. Similarly, the truth value of $(a[3], b[2])$ `TRIGGERS NEXT` c (that is, 3 occurrences of a followed by 2 occurrences of b must be followed by an occurrence of c) can be determined by considering a prefix of length 6. On the other hand, there does not exist a bound k such that the truth value of the formulas a `UNTIL` b or $(a[2]b^*c)$ `TRIGGERS NEXT` d (that is, 2 occurrences of a followed by some number of occurrences of b and then a c must be followed by d) can be determined by considering prefixes of length k. Suppose that P is a bounded ForSpec specification. Proving `ALWAYS` P by explicit induction requires the following:

(i) Manually find a strengthening Q of P for which `ALWAYS` Q implies `ALWAYS` P. Again, usually Q is chosen to be $P \wedge \langle\text{something}\rangle$.

(ii) Find an N for which the following two proofs are achievable:
   (a) Base: Prove that $M \models$ `ALWAYS`$[0, N]$ Q

(b) Step: Prove that $uninit(M) \models (\texttt{ALWAYS}[0, N]\ Q) \Rightarrow \texttt{NEXT}[N+1]\ Q$

For bounded formulas Q, both (ii-a) and (ii-b) can be proved using bounded model checking, as we discuss later. For readers not familiar with the ForSpec language, the formula $\texttt{ALWAYS}[0, N]\ Q$ means that Q holds in the first $N+1$ states of a path. Similarly, $\texttt{NEXT}[N+1]\ Q$ means that Q holds in the $N+2$-nd state of a path.

Now, one can easily see that the requirements (ii-a) and (ii-b) in explicit induction are the analogs of their counterparts in windowed induction. As a consequence the explicit induction is sound and complete too (recall the loopfreeness default constraint). The major difference between explicit induction and implicit induction is that the induction is no longer an internal algorithm inside the model checker. Rather, the induction scheme becomes an integral part of the specification, where it can be easily controlled by the user, using the expressiveness of the ForSpec specification language. This combines the qualities of windowed induction with the ability to prove properties that are more complex than simple invariants.

2 Tool Issues

This section examines the tool support necessary to implement the induction checks (ii-a) and (ii-b) described in Subsection 1.4. Note first that (ii-a) comes down to checking the assertion $\texttt{ALWAYS}\ Q$ until bound N (see discussion below for the impact of formula depth on the bound). This is a plain bounded model checking problem.

Similarly (ii-b) can be reduced to performing bounded model checking for the assertion $\texttt{ALWAYS}\ Q$ with bound exactly $N+1$ (again, see discussion below), with the assumption $\texttt{ALWAYS}[0, N]\ Q$, on an uninitialized version of M. This means that the tool performs bounded model checking at bound exactly $N+1$ on a model that is formed by composing three smaller models:

- the uninitialized model $uninit(M) = (S, S, T)$, derived from the original model $M = (S, S_0, T)$,
- the initialized automaton of the assertion $\texttt{ALWAYS}\ Q$,
- the initialized automaton of the assumption $\texttt{ALWAYS}[0, N]\ Q$.

While the model M has to be uninitialized for the induction to be sound, the two automata compiled by ForSpec have to be initialized as in a regular run. For instance, the counter used in the $[0, N]$ time window of the assumption needs to start from 0, otherwise our check does not implement correctly the inductive step.

To sum up, we have reduced (ii-b) to another instance of bounded model checking where the property $\texttt{ALWAYS}\ Q$ is checked exactly at bound $N+1$, and the initial constraints are built selectively only from the two ForSpec automata, but not from the model M itself, or from other ForSpec properties.

To solve this last issue, we offer to the user a new ForSpec keyword, `INDUCTION_HYPOTHESIS`, that he can use to mark the assumption `ALWAYS`$[0, N]\ Q$. This way, one can distinguish between assumptions that strengthen the induction proof and regular assumptions that specify the interface with neighbor RTL blocks. Based on the new keyword, the ForSpec compiler marks every one of the automata it generates as an assertion, assumption, or induction hypothesis. This information is passed to the model checker that makes sure to use only the initial constraints from the assertion and the induction hypothesis.

Finally, note that the bounds N and $N+1$ used in the two checks above are appropriate for the case where Q is a combinational property. We mentioned earlier that P (and hence Q) can be a bounded safety property. In this case, the bounds used depend on the length of the time windows employed in P. Usually, we end up choosing an offset k, such that the induction base (ii-a) is checked at bound $N + k$, while the induction step (ii-b) is checked at bound $N + k + 1$.

3 Usage Methodology

In this section we cover our methodology for working with explicit induction. From a usage point of view, there are three primary differences between explicit induction and implicit induction:

- The user must write the induction hypothesis explicitly, where the implicit induction builds it automatically from the assertions.
- The user must supply a window size in the explicit induction, while the implicit induction searches for a working induction window size by trying increasingly larger sizes.
- In implicit induction, the invariant is the result of automatic translation of the property. Hence, the user may find it difficult to strengthen the invariant (as explained above) and his most probable strategy would be to increase the window size. In explicit induction, the strengthening can be achieved by changing both the invariant and the window size.

The ForSpec directives needed to express explicit induction proofs are `ASSERT`, `ASSUME`, and `INDUCTION_HYPOTHESIS`. The keyword `ASSUME` is used to give auxiliary assumptions (e.g., assumptions about inputs etc.). The explicit induction technique does not depend on the usage of ForSpec, the ForSpec constructs described below help expressing and maintaining the induction hypotheses.

We exploit the "block template" construct in ForSpec to generate related formulas for a given temporal property Q. For instance, we define a block template mk_induction_specs(Q, N) that generates for a given property Q, and a window N the induction hypothesis and the assertion necessary for the proof:

mk_induction_specs(Q, N) := {

upto_cycle_NN := ALWAYS[$0, N$] Q;

at_all_times := ALWAYS Q;

}

The explicit induction directives for Q would then look as follows in ForSpec:

myspec := mk_induction_specs(Q, N); // instantiate the block

`INDUCTION_HYPOTHESIS` myspec/upto_cycle_N;

`ASSERT` myspec/at_all_times;

We then check these two directives in a model checking run of $N+1$ cycles. Not only does the block template give us a compact notation, it also keeps the low level formulas denoting assertions and inductive hypotheses synchronized. They refer to the same property Q and the same bound N, which prevents false positives.

In some cases it is convenient to use a different window size for different induction hypotheses. This provides useful insight into the pipeline/logic depth that each property depends on, and also helps select minimal sufficient model checker bounds to control complexity. It is a strength of this methodology that it is flexible enough to use either independent or identical induction window sizes for different specs.

For any property that can be proved inductively, there will be some induction window size for which it and all larger window sizes yield a successful proof, and all smaller window sizes fail with a counterexample in the formal model. Initially it is not known what window size is needed, so starting with larger window sizes can be beneficial. On the other hand, shorter window sizes find counterexamples more quickly, and shorter counterexamples are easier to debug.

As a general rule, we estimate an induction window size N that might work and try to prove the property Q. If we get a failure, we examine the counterexample. If the counterexample looks like it could be due to out-of-sync initialized pipelines, the induction window size needs to be increased. If the counterexample looks like it is caused by some state in the formal model that should not be reached in the actual circuit, the inductive hypothesis probably needs strengthening. The strengthened hypothesis typically adds a new constraint that forbids some problematic state combination that contributes to the counterexample.

Assertions that depend on simple pipelines demonstrate the advantage of windowed induction over "depipelining", i.e., of turning a windowed induction hypothesis into a larger invariant for simple induction. Where a windowed induction hypothesis can simply reference values of interest at the ends of the pipelines, a simple induction invariant must explicitly express a constraint at every pipeline stage. When the logic driving the assertion to prove is more complex than simple pipelines, the effort and cost to construct a simple induction invariant is much higher.

This methodology also allows an intermediate approach between pure simple induction and long induction windows. For simple pipelines amenable to depipelining, one can partition the pipeline into two (or more) pieces of roughly equal length, and express invariants at just the partition points and end. For very long pipelines this can roughly halve the required induction window size with only a small depipelining cost in invariant construction. It is a strength of any windowed induction technique, including ours, that this trade off is available.

4 Application to the Lock Protocol in the Pentium™4 Processor

The lock protocol is used in the Pentium™4 to allow different threads to execute atomic operations on several shared resources. The lock protocol is important to verify because it interacts subtly with several other microarchitectural features, making its functional correctness crucial. The lock protocol interacts with the cache coherence protocol, as well as with several other performance optimizations.

One basic requirement for such a protocol is mutual exclusion: no resource can be locked by two threads at once. This property is expressed by a ForSpec formula of the form `ALWAYS` a `TRIGGERS` $b[k]$, where a and b are certain signals of the design and k is an integer. Note that this property is bounded. Using bounded model checking this property was proved on all traces of up to 50 cycles. Given the importance of this property, it was necessary to get a full unbounded proof and we used explicit induction for that purpose.

When trying to prove the mutual exclusion property, we quickly get induction failures. The model checker provides a witness trace for which the inductive step does not hold. Usually this is due to starting in states that are unreachable in the real model. For instance, the control part of the protocol is modeled as a finite state machine. Some induction failures are traces that include concurrent occurrence of certain events that cannot actually happen together in the real model. Adding a simple induction hypothesis eliminates such trivial failures.

Developing the inductive proof is therefore an iterative process, where we keep strengthening the induction hypothesis based on the failures of previous attempts to establish induction. Overall, we had to add about twenty constraints before the hypothesis was sufficiently strong to establish induction. The window size of the inductive proofs differed from property to property. Only three properties were provable with a window size of one cycle. All the other properties were proved with a window size of six to twelve cycles.

The model we verified is quite large, containing about 12,000 state elements. This is considerably beyond the capacity (a few hundred state elements) of BDD-based model checkers. The verification effort for the proved properties described here took three to six person-months. Most induction-step runs completed within 20 minutes, checking 36 steps, using under 600M of memory. All induction-step runs completed within 3 hours, checking 48 steps, using under 1G memory.

5 Similar Approaches to Induction

Two potential alternatives invite a comparison with our approach. The most direct comparison is with the implicit induction based on [13]. This approach, as currently implemented in Thunder is incapable of inductively proving most temporal properties written in ForSpec, so no direct comparison is possible. We explained earlier the reasons for the failure of implicit induction for temporal properties. While the capacity problem cannot be eliminated by using a different compilation scheme, we believe that it can be alleviated. For example, were we to compile the properties into deterministic automata, the counterexample traces for the induction step (ii-b) would be more constrained, reducing the required window size N. This is a topic of further research. Even if that technology becomes available, it is possible that for very large models our approach would still be preferable to implicit induction, because of the insight into induction window lengths our approach can give on a property by property basis.

Another comparison with our approach is induction using STE instead of SAT as the bounded model checker. Induction with STE [12] has been used successfully at Intel for several years, particularly for datapath logic and floating-point property proofs. Sajid and Kaviola pioneered the extension of STE induction to a moderately complex control logic property [11]. Direct comparisons between the STE and SAT induction approaches are not easy because of tool differences. Our 12000 state element model, even reduced to the critical latches, is likely more than twice the size of the 3000 state element model in [11]. Perhaps the largest distinction between SAT induction and STE induction is the apparent difficulty or impracticality of doing windowed induction (vs. simple induction) with STE. Windowed induction using STE is theoretically possible, but it has problems with rapid variable blowup and/or *antecedent conflicts*. The work of [11] includes simple induction for exactly this reason.

6 Results

The concepts presented here have been implemented in the formal verification tool suite at Intel. Only minor modifications were required to the ForSpec compiler and to Thunder, our SAT based model checker. The explicit induction approach has been successfully used in the DPG Formal Verification Group. In particular, the most impressive application was the full verification of the lock protocol described above. The size of the model,12000 state elements, and the complexity of the protocol speak for themselves. But beyond the quantitative data, there is the impact of a new methodology that can address verification problems that cannot be handled with the existing technologies.

The use of windowed induction seems to be a critical factor in enabling successful proofs of these properties. If it was necessary to *depipeline* our windowed induction hypotheses into hypotheses sufficient for simple induction, a reasonable estimate is that spec volume would increase by at least a factor of 10, effort at least triple, and comprehensibility and maintainability would be considerably reduced.

7 Summary

The methodology and tool support presented in this paper address the challenge of fully verifying complex temporal properties on large RTL designs. The methodology advocated here is interactive development of inductive proofs. There is much ongoing research for automating induction proofs, but no satisfactory technique has been found, even for combinational properties. We believe that for proofs of the complexity encountered in our work, user guidance is needed for finding the correct induction invariants.

At the core of our approach is the explicit induction, a new induction scheme targeted to temporal properties, and to interactive development of inductive proofs. The case of temporal properties was not adequately addressed by previously known methods. The innovative idea in the explicit induction is to make the induction scheme an explicit part of the specification, where it can be easily controlled, using a highly expressive language like ForSpec.

The current experience shows that the explicit induction is capable of handling verification problems that were previously intractable with all the existing technologies. To further push this approach, our future work will focus on automating some of the manual tasks, for instance by having the tool suggest candidates for induction invariants.

References

[1] Beer, I., S. Ben-David, C. Eisner and A. Landver, *RuleBase: An industry-oriented formal verification tool*, in: *Proc. 33rd Conference on Design Automation* (1996), pp. 655–660.

[2] Biere, A., A. Cimatti, E. Clarke, M. Fujita and Y. Zhu, *Symbolic model checking using SAT procedures instead of BDDs*, in: *Proc. 36th Design Automation Conference* (1999), pp. 317–320.

[3] Biere, A., E. Clarke, R. Raimi and Y. Zhu, *Verifying safety properties of a PowerPC[tm] microprocessor using symbolic model checking without BDDs*, in: *Computer Aided Verification, Proc. 11th International Conference*, Lecture Notes in Computer Science **1633** (1999), pp. 172–183.

[4] Bryant, R., *Graph-based algorithms for boolean-function manipulation*, IEEE Trans. on Computers **C-35** (1986).

[5] Claessen, K., *Induction and state machines* (1999), unpublished.

[6] Clarke, E., O. Grumberg and D. Peled, "Model Checking," MIT Press, 1999.

[7] Copty, F., L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella and M. Vardi, *Benefits of bounded model checking at an industrial setting*, in: *Computer Aided Verification, Proc. 13th International Conference*, Lecture Notes in Computer Science **2102** (2001), pp. 436–453.

[8] Kupferman, O. and M. Vardi, *Model checking of safety properties*, Formal methods in System Design **19** (2001), pp. 291–314.

[9] Manna, Z. and A. Pnueli, "The Temporal Logic of Reactive and Concurrent Systems: Safety," Springer-Verlag, New York, 1995.

[10] McMillan, K., "Symbolic Model Checking," Kluwer Academic Publishers, 1993.

[11] Sajid, K. and R. Kaviola, *Verification of pentium™4 BUS recycle logic using symbolic simulation and induction*, in: *Intel Design Test and Technology Conference*, 2003.

[12] Seger, C. and R. Bryant, *Formal verification by symbolic evaluation of partially-ordered trajectories*, Formal Methods in System Design **6** (1995), pp. 147–189.

[13] Sheeran, M., S. Singh and G. Stalmarck, *Check safety properties using induction and a SAT-solver*, in: *Proc. 3rd Conference on Formal Methods in Computer-Aided Design*, Lecture Notes in Computer Science **1954** (2000), pp. 108–125.

[14] Vardi, M. and P. Wolper, *Reasoning about infinite computations*, Information and Computation **115** (1994), pp. 1–37.