Checking Safety Properties Using Induction and a SAT-Solver

Mary Sheeran[1,2], Satnam Singh[3], and Gunnar Stålmarck[1,2]

[1] Prover Technology AB, Alströmergatan 22, 2tr, SE-112 47 Stockholm, Sweden
gunnar@prover.com
[2] Chalmers University of Technology, SE-412 96, Göteborg, Sweden
ms@prover.com
[3] Xilinx Inc., 2100 Logic Drive, San Jose, California CA95124, USA
Satnam.Singh@xilinx.com

Abstract. We take a fresh look at the problem of how to check safety properties of finite state machines. We are particularly interested in checking safety properties with the help of a SAT-solver. We describe some novel induction-based methods, and show how they are related to more standard fixpoint algorithms for invariance checking. We also present preliminary experimental results in the verification of FPGA cores. This demonstrates the practicality of combining a SAT-solver with induction for safety property checking of hardware in a real design flow.

1 Introduction

We are interested in the problem of checking safety properties of large finite state machines using a SAT-solver. This has become an important research topic in recent years, and a number of apparently different approaches have been proposed [1, 2, 5, 7, 12]. Several of these methods seem promising, and experimental work to evaluate them is being carried out. We also need to develop a greater understanding of the problem and its various solutions in a more abstract sense. This paper contributes to this ongoing work in two ways. First, we explain some induction-based methods of safety property checking. Although applications of some of these methods have been reported [9], the methods themselves have not been properly documented in the literature. This paper attempts to remedy this. Second, we demonstrate the practicality of the approach by giving experimental results for the verification of real FPGA cores at Xilinx, Inc.

2 The Problem that We Would Like to Solve

Given a finite state machine M with initial states satisfying I and state transition relation T, we would like to check whether or not a property P holds for all reachable states. The transition relation T is a binary relation on the set of states S. We call a state that satisfies P a P-state, and a system in which all reachable states are also P-states is called P-safe. The reachable states are those that can be reached by T-transitions starting from an initial state.

© Springer-Verlag Berlin Heidelberg 2000

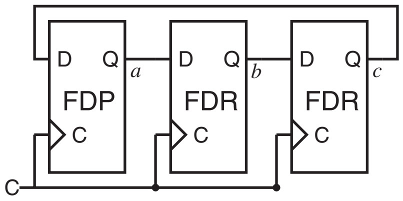

Fig. 1. A 3-bit ring counter

Example 1. Figure 1 shows a circuit for a 3-bit ring counter which has the property that only one bit is high at any given moment. When this circuit is reset into its initial state the Q output of the FDP flip-flop is set to 1 and the Q outputs of the two FDR flip-flops are set to 0. (We equate 0 with $False$ and 1 with $True$.) This circuit has no inputs except for the clock C. Figure 2 shows a state transition diagram for this circuit which has one initial state $(1, 0, 0)$. In general, the state will be a finite vector of boolean variables. Transitions are

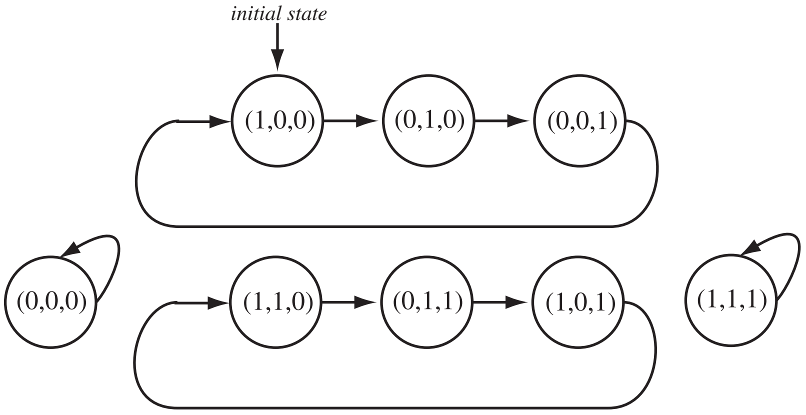

Fig. 2. State transition diagram for a 3-bit ring counter

shown as arrows between states. So, the circuit cycles between three reachable states. Let us call the three boolean state variables (a, b, c) as shown in Figure 1. Then, the property that only one bit should be high is represented by the formula $(a \oplus b \oplus c) \wedge \neg(a \wedge b \wedge c)$, informally "an odd-number of bits should be high, but not all three", which we call *oneHigh*. ($\oplus$ stands for exclusive or.) This property holds for all of the reachable states (on the top row of the diagram) and so the system shown is *oneHigh*-safe. An example of a property that does not hold for all reachable states is the formula $\neg c$. It holds for the initial state and for its successor, but not for the following state, so a suitable countermodel to the assertion that $\neg c$ holds for all reachable states is the sequence of states $(1,0,0),(0,1,0),(0,0,1)$.

Generally such an error trace is a possible sequence of states starting at the initial state, in which all but the last state satisfy the required condition. When we check systems for P-safety, we would like to generate such a trace when the system turns out not to be P-safe. The question of how to get from a system description to a transition relation is not considered here. For an introduction to model checking in general, see reference [6].

2.1 Transition Relations and Paths

In order to be able to formulate the problem more precisely, we introduce notation for various types of paths through the graph of a transition relation. We write $T(x,y)$ to indicate that x is related to y by the transition relation T. Let us assume for notational convenience that the transition relation being examined is always T. For example the 3-bit ring counter presented in the previous section has a transition relation which relates the current state (a,b,c) to the next state (a',b',c') such at $a'=c$, $b'=a$ and $c'=b$. Now we define what it means for a sequence of states to be a path through T.

$$\mathit{path}(s_{[0..n]}) \;\hat{=}\; \bigwedge_{0\le i<n} T(s_i, s_{i+1})$$

Read $\hat{=}$ as “is defined to be”. $s_{[0..n]}$ is shorthand for the sequence of state $(s_0, s_1, \ldots, s_n)$. We say that a path has length n if it makes n T-transitions. A path of length zero contains a single point and makes no transitions. To assert that a property Q holds of every point in a path, we write $\mathit{all}.Q(s_{[0..n]})$.

Later, we will have reason to restrict the paths in such a repeated composition to be *loop free*, so that every element of the path is distinct. We define

$$\mathit{loopFree}(s_{[0..n]}) \;\hat{=}\; \mathit{path}(s_{[0..n]}) \wedge \bigwedge_{0\le i<j\le n} s_i \ne s_j$$

The concatenation of two loop-free paths is not necessarily loop-free, but the sub-paths of a loop-free path are themselves loop-free. Thus

$$\mathit{loopFree}(s_{[0..(i+j)]}) \;\rightarrow\; \mathit{loopFree}(s_{[0..i]}) \wedge \mathit{loopFree}(s_{[i..(i+j)]}) \qquad (1)$$

Sometimes we only want to talk about the ends of paths. So we are happy to view *path*, for instance, not only as a predicate on paths but also as a binary relation on points. We write $\mathit{path}_i(s_0, s_i)$ to indicate that there is a path from s_0 to s_i through i copies of T. This corresponds to quantifying away the internal points. So, for example

$$\mathit{path}_n(s_0, s_n) \;\leftrightarrow\; \exists s_1 \ldots s_{n-1}.\, \mathit{path}(s_{[0..n]}) \qquad (2)$$

Finally, we define what it means for a path to be a *shortest* path. In this case, we are only interested in the ends of paths. A path from a to b is shortest if it joins a and b and if a and b are not joined by any shorter path. Define

$$\mathit{shortest}(s_{[0..n]}) \;\hat{=}\; \mathit{path}(s_{[0..n]}) \;\wedge\; \neg(\bigvee_{0 \leq i < n} \mathit{path}_i(s_0, s_n)) \tag{3}$$

Shortest paths are also loop-free. Note that the definition of *shortest* in fact contains many existential quantifiers, because we have repeatedly used $\mathit{path}_i(s_0, s_n)$. For a finite transition relation T, there exists a largest k for which $\mathit{shortest}(s_{[0..k]})$ holds for some sequence of states. In other words, there is a longest shortest path in a finite state transition graph. The length of that path is usually called the *diameter* of the graph. The diameter of the state transition graph shown in figure 2 is 2.

2.2 Formulating the Problem

Let T be a transition relation on the set of states S. We assume that the domain of T is the entire set of states S, so that every state has a successor through T. Let I characterise the initial states, and P the property of states that we want to check.

We want to show that starting in an initial state and repeatedly applying the transition relation always leads to a state satisfying P. That is, we want to prove

$$\forall i. \forall s_0 \ldots s_i. \;\; (I(s_0) \;\wedge\; \mathit{path}(s_{[0..i]}) \;\rightarrow\; P(s_i))$$

where $i \geq 0$ and the s_i range over states. Or we can work backwards from the bad states. We want to show that starting in a state violating P and working backwards through T always leads to a non-initial state, that is

$$\forall i. \forall s_0 \ldots s_i. \;\; (\neg I(s_0) \;\leftarrow\; \mathit{path}(s_{[0..i]}) \wedge \neg P(s_i))$$

Both of these turn out to be the same thing as proving

$$\forall i. \forall s_0 \ldots s_i. \;\; \neg(I(s_0) \;\wedge\; \mathit{path}(s_{[0..i]}) \;\wedge\; \neg P(s_i))$$

This gives a more symmetrical view of the problem. In words, we want to show that there are no paths that start in an initial state and end in a non-P-state.

3 A First Solution

How can we divide our problem up into smaller sub-problems?

A possible first solution is to check that

$$\forall s_0 \ldots s_i. \;\; \neg(I(s_0) \;\wedge\; \mathit{path}(s_{[0..i]}) \;\wedge\; \neg P(s_i)) \tag{4}$$

holds for $i = 0$, $i = 1$, $i = 2$, and so on. This corresponds to checking that $I(s_0) \wedge \mathit{path}(s_{[0..i]}) \wedge \neg P(s_i)$ is contradictory (or that $\neg(I(s_0) \wedge \mathit{path}(s_{[0..i]}) \wedge \neg P(s_i))$ is a tautology) for each i, for arbitrary s_0 to s_i. If the property is violated somewhere in the reachable states, we will eventually find an i for which $I(s_0) \wedge \mathit{path}(s_{[0..i]}) \wedge \neg P(s_i)$ is satisfiable. Then, we know that there is a path of length i from an initial state to one violating P, and indeed the assignment of values to $s_{[0..i]}$ that makes the formula satisfiable is such a path, and can be used for debugging purposes. Also, we know that there is no shorter such error-trace. For the special case of simple safety properties, Bounded Model Checking, a particular form of model checking based on SAT-solving proposed by Clarke and his collaborators [2], reduces to a similar kind of iteration and satisfiability check.

If the system is P-safe, formula (4) will always hold. The question is how do we know when it is safe to stop incrementing i and conclude that the system is P-safe? It is no good waiting for $I(s_0) \wedge \mathit{path}(s_{[0..i]})$ to be contradictory, say. Given that there is an initial state, this will never happen, as we assume that every state has a successor through T, so there are always loops in both the reachable and the unreachable state space.

A better strategy is to stop when $I(s_0) \wedge \mathit{loopFree}(s_{[0..i]})$ becomes contradictory. Then, we stop when we have checked every loop-free path (and thus every state) in the reachable states. We can then safely conclude that the system is P-safe. Similarly, we can keep checking until $\mathit{loopFree}(s_{[0..i]}) \wedge \neg P(s_i)$ becomes contradictory, and stop, again with a positive answer, when we have checked all states reachable backwards from those violating P. This solution is given in pseudo-code below (Algorithm 1). The function Sat corresponds to a call to a SAT-solver. The function Sat takes an expression and returns True if there exists an assignment to the variables (in this case $s_{[0..i]}$) which make the whole expression true. Here, the trace $c_{[0..i]}$ is an assignment to the variables $s_{[0..i]}$ that

Algorithm 1 First algorithm to check if system is P-safe

```
$i$=0
**while** True **do**
  **if** not Sat($I(s_0) \wedge \mathit{loopFree}(s_{[0..i]})$) or not Sat(($\mathit{loopFree}(s_{[0..i]}) \wedge \neg P(s_i)$)) **then**
    return True
  **end if**
  **if** Sat($I(s_0) \wedge \mathit{path}(s_{[0..i]}) \wedge \neg P(s_i)$) **then**
    return Trace $c_{[0..i]}$
  **end if**
  $i = i + 1$
**end while**
```

makes $I(s_0) \wedge \mathit{path}(s_{[0..i]}) \wedge \neg P(s_i)$ true, and so is a suitable error trace.

Let us consider the case when the answer is True. We prove that for all i

$$I(s_0) \wedge \mathit{loopFree}(s_{[0..i]}) \wedge \neg P(s_i)$$

is contradictory. That is, we show that there is no loop-free path, starting from the initial state, in which the final state violates P. If there is no such path, then the system must be P-safe, and thus the method is sound. When the answer is True, we know from the condition in the first if statement that for some smallest k one of $I(s_0) \wedge \mathit{loopFree}(s_{[0..k]}))$ and $\mathit{loopFree}(s_{[0..k]}) \wedge \neg P(s_k)$ must be contradictory (or not satisfiable). We also know, from the second if statement, that for $i < k$ it is the case that

$$I(s_0) \wedge \mathit{path}(s_{[0..i]}) \wedge \neg P(s_i)$$

is contradictory. A consequence is that for $i < k$

$$I(s_0) \wedge \mathit{loopFree}(s_{[0..i]}) \wedge \neg P(s_i) \tag{5}$$

is also contradictory. If there are no paths linking an initial state to a non-P-state, then there are no loop-free paths doing so either.

It remains to be shown that equation (5) holds for $i \geq k$. For $i \geq k$, we know (by equations 1 and 2 and some quantifier manipulation) that

$$\begin{aligned}&I(s_0) \wedge \mathit{loopFree}(s_{[0..i]}) \wedge \neg P(s_i)\\ &\qquad \rightarrow I(s_0) \wedge \mathit{loopFree}(s_{[0..k]}) \wedge \mathit{loopFree}(s_{[k..i]}) \wedge \neg P(s_i)\end{aligned}$$

and that

$$\begin{aligned}&I(s_0) \wedge \mathit{loopFree}(s_{[0..i]}) \wedge \neg P(s_i)\\ &\qquad \rightarrow I(s_0) \wedge \mathit{loopFree}(s_{[0..m]}) \wedge \mathit{loopFree}(s_{[m..i]}) \wedge \neg P(s_i)\end{aligned}$$

for some m. But at least one of those right hand sides is unsatisfiable since one of $I(s_0) \wedge \mathit{loopFree}(s_{[0..k]})$ and $\mathit{loopFree}(s_{[0..k]}) \wedge \neg P(s_k)$ is. We conclude that for all i

$$I(s_0) \wedge \mathit{loopFree}(s_{[0..i]}) \wedge \neg P(s_i)$$

is contradictory. Thus a True answer does indeed indicate that the system is P-safe. The method is also complete. It returns True if the system is P-safe, and an error trace if not. The restriction to loop-free paths is necessary for completeness.

This algorithm is in a sense bidirectional; it can be thought of as working both forwards from the initial state and backwards from the bad states at the same time. Indeed, it is pleasingly symmetrical. We could swap the initial and the bad states, and replace the transition relation by its converse, and still get the same algorithm.

In the ring counter shown in figure 2, using this algorithm to check that *oneHigh* holds of all reachable states returns True when i becomes 3 since 2 is the length of the longest loop-free path starting from the initial state, and of the longest loop-free path ending in a non-*oneHigh* state. Checking for the formula $\neg c$ that we considered earlier returns the sequence of states $(1,0,0),(0,1,0),(0,0,1)$ as the assignment $s_0 = (1,0,0), s_1 = (0,1,0), s_2 = (0,0,1)$ is a satisfying assignment for the formula $I(s_0) \wedge \mathit{path}(s_{[0..i]}) \wedge \neg P(s_i)$ in this case.

4 Improving on this Solution

How can we improve this algorithm, bearing in mind that we will use a SAT-solver to check the formulas? Well, we can make the two termination conditions a bit tighter.

Let us think operationally for a moment, and imagine traversing the state transition graph. In the forward direction, we don't want to go back into an initial state as we would then be considering a longer path than necessary. We could in that case consider only the end part of the path starting from the second point that is an initial state. The original termination condition was $I(s_0) \wedge \mathit{loopFree}(s_{[0..i]})$ and now we want to replace it by

$$I(s_0) \wedge \mathit{all}.\neg I(s_{[1..i]}) \wedge \mathit{loopFree}(s_{[0..i]})$$

(In the special case where there is only one initial state, then this change is unnecessary, as the restriction to proper paths prevents us from returning to the initial state.) Similarly, in the backwards direction, we are uninterested in paths that have a non-P-state somewhere in the middle. We only want to consider paths in which all but the last state satisfy P. The new termination condition is then

$$\mathit{loopFree}(s_{[0..i]}) \wedge \mathit{all}.P(s_{[0..(i-1)]}) \wedge \neg P(s_i)$$

The resulting algorithm is given as Algorithm 2.

Algorithm 2 An improved algorithm to check if system is P-safe

```
$i$=0
while True do
  if not Sat($I(s_0) \wedge all.\neg I(s_{[1..i]}) \wedge loopFree(s_{[0..i]})$)
  or not Sat(($loopFree(s_{[0..i]}) \wedge all.P(s_{[0..(i-1)]}) \wedge \neg P(s_i)$)) then
    return True
  end if
  if Sat($I(s_0) \wedge path(s_{[0..i]}) \wedge \neg P(s_i)$) then
    return Trace $c_{[0..i]}$
  end if
  $i = i + 1$
end while
```

One of us was sorely tempted to make use of facts proved in earlier iterations to make further restrictions in both termination conditions, restoring a pleasing symmetry. But this turns out to be a bad idea in practice because of the need to rely on previous iterations. When a circuit requires a very high induction depth to prove a property, it is simply too expensive to iterate all the way up to that depth, from zero. The proofs that find that we cannot yet terminate are much slower than the successful proofs of termination conditions. So, we should instead concentrate on removing the need to iterate upwards from zero depth!

We need to change the check for bad paths so that it can find bad paths of length 0 up to i, and not just of length exactly i. It turns out to be convenient to switch the order of the check for bad paths and the check for termination.

Algorithm 3 An algorithm that need not iterate from 0

```
$i =$ some constant which can be greater than zero
**while** True **do**
  **if** Sat($I(s_0) \wedge \mathit{path}(s_{[0..i]}) \wedge \neg \mathit{all}.P(s_{[0..i]})$) **then**
    return Trace $c_{[0..i]}$
  **end if**
  **if** not Sat($I(s_0) \wedge \mathit{all}.\neg I(s_{[1..(i+1)]}) \wedge \mathit{loopFree}(s_{[0..(i+1)]})$)
  or not Sat($(\mathit{loopFree}(s_{[0..(i+1)]}) \wedge \mathit{all}.P(s_{[0..i]}) \wedge \neg P(s_{i+1})$) **then**
    return True
  **end if**
  $i = i + 1$
**end while**
```

Now, we are no longer obliged to iterate all the way up from zero, as we have removed the dependence between iterations. The length of the longest serial connection of latches (or other delay elements) is usually a lower bound on the number of iterations needed, so that is a good starting point. The algorithm is still sound, even if we start at “too high” a value of i. In that case, though, the error trace returned is no longer guaranteed to be of minimal length. Each iteration also begins to look more like an inductive proof. Rewriting some of the subproblems to equivalent ones gives Algorithm 4. Here the call to Taut invokes a SAT-solver to establish whether its argument expression is always true.

Algorithm 4 A forwards version of the algorithm

```
$i =$ some constant which can be greater than zero
**while** True **do**
  **if** Sat($\neg(I(s_0) \wedge \mathit{path}(s_{[0..i]}) \rightarrow \mathit{all}.P(s_{[0..i]}))$) **then**
    return Trace $c_{[0..i]}$
  **end if**
  **if** Taut($\neg I(s_0) \leftarrow \mathit{all}.\neg I(s_{[1..(i+1)]}) \wedge \mathit{loopFree}(s_{[0..(i+1)]})$)
  or Taut($(\mathit{loopFree}(s_{[0..(i+1)]}) \wedge \mathit{all}.P(s_{[0..i]}) \rightarrow P(s_{i+1})$) **then**
    return True
  **end if**
  $i = i + 1$
**end while**
```

Now we can begin to see the inductive shape of the proof. The first if statement is the base case. It checks that P holds in the first $i+1$ states. The second disjunct in the condition in the next if statement checks that after $i+1$ P-states in a row one is guaranteed to reach another P-state. By induction, we conclude that every loop-free path starting at the initial state contains only P-states. We call this *strengthened induction with depth* i, and it proves that the system is P-safe. The word *strengthened* refers to the restriction to loop-free paths, which is an additional constraint on the unrollings of the transition relation. Without this constraint, induction with depth gives an incomplete method. Later, we shall see a stronger variant of induction. However, even this the weakest form of induction works well for some kinds of hardware verification, and our experimental results from core verification use only this form of induction so far. We don't yet know whether or not the stronger forms of induction are useful in practice. Returning to the remaining disjunct in the second if statement, it checks whether or not we can stop because all of the loop-free paths in the reachable states have already been checked (by the base cases). This view of the algorithm has more of a left-to-right character than the original one, but the two algorithms are in fact the same! Indeed, there is also a right-to-left version, which you can get by replacing the condition in the first if statement by $all.\neg I(s_{[0..i]}) \leftarrow path(s_{[0..i]}) \wedge \neg P(s_i)$. Now, view this as the base case of the induction, and the first disjunct of the second if statement as the step. In fact, though, both versions behave identically as the condition that we have just introduced traps exactly the same bad paths as the previous one. It is in the left-to-right form that we have presented the algorithm earlier[12]. We feel that the more symmetrical presentation given here is enlightening. In particular, it has helped us to understand the importance of the double termination check which gives us an algorithm that can be seen as working both forwards and backwards.

Returning to the ring counter example and again checking the *oneHigh* property, we find that the tighter termination condition now allows us to terminate when i is 0. This is because there are no paths of length 1 connecting a *oneHigh* state to a non-*oneHigh* state. If the property being checked happens to coincide exactly with the reachable states, as in this example, then there are no paths from a P-state to a non-P state, so induction with depth 0, which is just ordinary induction, will succeed. Restricting the backwards termination condition to consider only paths in which all but the last state satisfy P is an important refinement of the algorithm.

The limiting factor for this algorithm is the cost of adding the extra constraints in the termination conditions that constrain the paths to be loop-free. This needs in the order of n^2 inequalities between states for path length n. Since Stålmarck's method copes well with very large formulas, the limit is not as constraining as it might appear [11]. Somewhat surprisingly, most of the examples that we have tried so far have needed relatively low induction depths. However, there will clearly be deep systems that we simply cannot cope with.

The algorithms that we have presented so far are suited for use with a SAT-solver. The formulas that need to be checked are given by the definitions of *path*, *loopFree* and *all.P*.

At Prover Technology AB, the algorithm is implemented in the prototype LUCIFER tool for checking safety properties of Lustre programs [10]. LUCIFER has been used in a number of industrial verification projects at Prover Technology

AB. It is currently being evaluated at a large aerospace company, for use in the verification of control programs and a product incorporating these algorithms and others is being developed. LUCIFER is also in use in a project to develop a design flow incorporating formal verification for FPGA cores at Xilinx, Inc. [9]. Later, in section 7, we present experimental results from this project.

5 A Second Stronger Solution

The algorithm that we have just presented has the advantage that it can be implemented using a plain SAT-solver. However, it is unsatisfactory when one considers the necessary number of iterations before it terminates, for a P-safe system. The number of iterations required is either the length of the longest loop-free path starting from an initial state (and proceeding through non-initial states), or the length of the longest loop-free path consisting of all P states followed by a non-P-state, whichever is the shorter. But this could easily be far too many iterations! The longest loop-free path between a pair of states may be *much* longer than the shortest path between them, so the algorithm may needlessly consider long paths. We would like to consider only *shortest* paths between pairs of states. This insight leads to a new solution. The adaption of the algorithm to consider only shortest paths is straightforward. Everywhere we had *loopFree*, we substitute *shortest*. The reasoning is just as before, except that

Algorithm 5 A version of the algorithm that considers shortest paths

```
$i$= some constant which can be greater than zero
**while** True **do**
  **if** Sat($I(s_0) \wedge \mathit{path}(s_{[0..i]}) \wedge \neg \mathit{all}.P(s_{[0..i]})$) **then**
    return Trace $c_{[0..i]}$
  **end if**
  **if** not Sat($I(s_0) \wedge \mathit{all}.\neg I(s_{[1..(i+1)]}) \wedge \mathit{shortest}(s_{[0..(i+1)]})$)
  or not Sat($(\mathit{shortest}(s_{[0..(i+1)]}) \wedge \mathit{all}.P(s_{[0..i]}) \wedge \neg P(s_{i+1})$) **then**
    return True
  **end if**
  $i = i + 1$
**end while**
```

this time we prove that for all i

$$I(s_0) \wedge \mathit{shortest}(s_{[0..i]}) \wedge \neg P(s_i)$$

is contradictory. Considering only shortest paths does not reduce the set of states considered, so this condition still captures P-safety. The big difference is that for a P-safe system, this algorithm may terminate much earlier.

Note, however, that using *shortest* in the algorithm means that we have introduced many existential quantifiers. The algorithm is no longer suitable for use with a plain SAT-solver, but needs quantifier elimination. Work on the FixIt tool shows that one can get quite far in SAT-based reachability analysis using relatively simple quantifier elimination [1]. Another alternative is to use the QBF-specific part of Stålmarck's translation of finite domain many sorted first order logic into propositional logic [13]. We will not consider either option further in this paper, but note that much experimental work remains to be done.

We call the *forward diameter* of a graph the length of the longest shortest path starting in an initial state and proceeding through non-initial states, and the *backward diameter* the length of the longest shortest path consisting of all P-states followed by a non-P-state. The number of iterations needed for a P-safe system is the minimum of the forward and backward diameters.

Algorithm 5 still considers entire paths. We can think of modifying it so that it instead only considers the two end points of a path. This makes sense as we are now considering shortest paths. Having made that step, we can make one last quantifier elimination in each of the termination conditions. We would rather not be forced to iterate very far by a system that has a very long shortest path from an initial state to x, if x is reachable much earlier from a *different* initial state. We would somehow like to bundle all the initial states together. Our definition of *shortest* gives us shortest paths from a single state. Now we would like to consider shortest paths from a set of states. We modify the definition of *shortest*. S characterises the set of states.

$$\mathit{shortest}'_n(S,b) \;\hat{=}\; \exists a.\,(S(a) \wedge \mathit{path}_n(a,b)) \wedge \neg\exists a.\,(S(a) \wedge \bigvee_{0\le i<n} \mathit{path}_i(a,b))$$

Now, $\mathit{shortest}'_n(S,b)$ also characterises a set of states. The important point to note is that we have added two further existential quantifiers. Similarly, we will write $\mathit{shortest}'_n(a,S)$ for the assertion that there is a shortest path from state a into the set characterised by S. For simplicity, we also omit the constraints on internal points on paths. The result is shown as Algorithm 6.

Algorithm 6 A set-based version of the algorithm that considers shortest paths

```
$i=$ some constant which can be greater than zero
**while** True **do**
  **if** Sat($I(s_0) \wedge \mathit{path}(s_{[0..i]}) \wedge \neg \mathit{all.P}(s_{[0..i]})$) **then**
    return Trace $c_{[0..i]}$
  **end if**
  **if** not Sat($\mathit{shortest}'_{i+1}(I, s_{i+1})$)
  or not Sat(($\mathit{shortest}'_{i+1}(s_0, \neg P)$) **then**
    return True
  **end if**
  $i = i + 1$
**end while**
```

This algorithm is similar to the standard method of checking safety properties using fixpointing and a simultaneous forward and backward analysis. The existential quantifiers that we have just added in the termination conditions have moved us from a path-based algorithm to one that operates on sets of states.

So, we have seen two kinds of solution, one that works directly with quantifier free formulas but that may need to iterate too far in practice in some cases, and a much stronger solution that seems to demand quantifier elimination. In between these two extremes, there is a range of solutions that can be explored. We want to look at various versions of constrained iteration of relations between *loopFree* and *shortest*.

One way to think about this range is to consider that the constraints that we apply are of the form "paths of length j or greater do not have any paths that are shorter than j between their end points". Now, choosing j to be 1 gives us the constraint that we used in our purely SAT-based solutions. We constrain all paths of length 1 or greater to have unequal end points (so that they obey $\neg path_j(x, x)$). If on the other hand, we choose j to be the length n of the path that we are considering, then we get the strongest form of induction. We demand that the entire path be a shortest path, so that its end points are not joined by any paths of length less than n. In between these two extremes, we can choose different values of j, to give a range of constraints. We have seen that setting j to be one gives loop-free paths. Choosing $j = 2$ gives what we call locally shortest paths; the consequence is that all paths of length 2 or greater obey the negation of the transition relation T and the entire path has unequal end points. This again is a constraint that can be expressed in pure propositional logic, and it is strictly stronger than the *loopFree* constraint. All of these constrained forms of iteration can be plugged into our basic algorithm. As yet, our experiments have been restricted to induction strengthened by the restriction to loop-free paths.

6 Related Work

Deharbe and Moreira have suggested using induction (with depth one) to check invariant properties of transition systems [8]. They modify a standard model checking algorithm to use induction for properties of the form $AG\ p$ (informally, "p is globally true along all paths"). Thus, this work is done in a context in which sets of states and image computations are expressed using BDDs. The authors point out that their method is incomplete and do not, to our knowledge, consider additional path constraints as we do.

In Bounded Model Checking (BMC), the user specifies a number of time steps, k, for searching from initial states for countermodels to properties [2, 3]. This work has alerted many to the possibilities of SAT-solvers in model checking. Reference [4] concentrates particularly on safety properties. It applies the method to the checking of safety properties of a PowerPC microprocessor at Motorola. The method used is either to search for a finite-length counter-example, exactly as we do, or to prove that the property is an inductive invariant, using induction with depth zero, that is ordinary induction. Induction with depth seems not to be considered. The authors point out that the technique is not complete. We know, however, that the authors considered restrictions to loop-free paths.

When it comes to termination conditions, the original BMC paper proposes to use the graph diameter as the length of the longest necessary unrolling. This would correspond, in our formulation, to terminating when $shortest(s_{[0..i]})$ becomes unsatisfiable. Our termination conditions are rather more refined in that they take account of the initial states and of the property to restrict attention only to more relevant sub-graphs. The restrictions to paths of the form "I then all not I" or "all P then not P" turn out to be important in practice, though we must admit that we have tried them mostly in the context of the weaker restrictions to loop-free paths.

For an interesting future research direction on the theme of variations on induction and SAT-solving, the reader is referred to recent work by Bjesse and Claessen (in this volume) on improving induction using a method first proposed by van Eijk [5].

Many of the methods that we propose here are already in use in automatic test pattern generation (ATPG) and one of our next steps will be to study SAT-based ATPG.

7 Results from FPGA Core Verification

At Xilinx the LUCIFER tool has been used to help verify the correctness of FPGA circuit cores – intellectual property that Xilinx distributes or sells. It is particularly important to ensure the correctness of these cores since users expect intellectual property to have undergone rigourous testing.

In a recent project many of the basic building block circuits, called BaseBLOXs, were verified with LUCIFER. Typical components include bus multiplexors, adders, subtractors, accumulators, comparators, complementors and counters. Full details of the BaseBLOX components are available from Xilinx [14].

Although in principle these circuits may not seem terribly challenging for formal verification, industrial versions include many extra inputs and configuration information which makes these circuits harder to verify. For example, the counter could have its count direction changed dynamically, it could also have a new value loaded dynamically on any clock tick, the amount to increment the count by can also be dynamically altered and two outputs indicate when certain count thresholds have been reached. Furthermore, the counter can have a clock enable as well as synchronous or asynchronous clears and sets to dynamically determined values. These factors inflate the number of primary inputs and outputs and increase the number of state elements.

The LUCIFER system accepts input in the Lustre language and generates a proof log which contains a countermodel if one exists plus other statistics about the verification. A system called Argus [9] was developed (see Figure 3) which translates Xilinx circuit designs in the EDIF netlist format into behaviourally equivalent Lustre. The Argus system then poses a question to the LUCIFER system to perform equivalence checking for the two input designs. When a counter model is found it is read from the generated proof log and translated into a simulation script for the ModelSim VHDL/Verilog simulator.

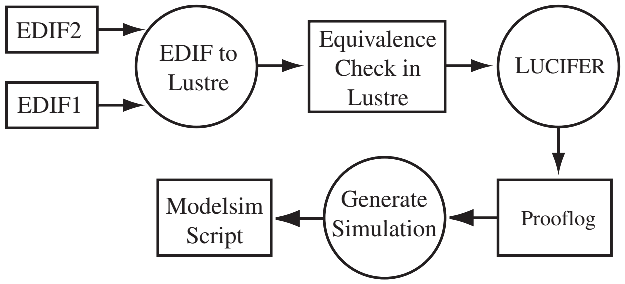

Fig. 3. Architecture of the Argus EDIF netlist equivalence checker

The core verification flow at Xilinx is shown in Figure 4. The Core Generator system is called upon to generate a specific instance of some core e.g. by specifying the size of a multiplier or the degree of pipelining required. The outputs of the Core Generator are (*i*) a highly optimised EDIF circuit netlist suitable for implementation on an FPGA and (*ii*) a VHDL behavioural description which should faithfully model the behaviour of the circuit contained in the EDIF netlist. This may easily not be the case since the behavioural descriptions are developed entirely independently of the optimised implementation circuits. The verification system that we have put in place performs equivalence checking to ensure that for specific instances of cores the implementation netlists are correctly modeled by the behavioural descriptions.

The flow involves taking a behavioural VHDL specification of a core's behaviour and synthesising a particular instance of it to produce an EDIF netlist. This acts as the specification of the required behaviour. Then the actual highly optimised structural VHDL (or other language) code for the corresponding core implementation is also elaborated into an EDIF file. The Argus equivalence checker then processes these two EDIF files. If the system finds a counter-model, then a simulation script identifying the sequence of events that led to the discrepancy is produced. This script can be used by the core designer or verifier from the ModelSim VHDL simulator to help identify the source of the problem.

A key aspect of our flow is that the verification engineer or core developer does not need to learn about any formal logic since the safety property that we wish to check (that the two circuits under examination always have the same output) is automatically generated by our system. Furthermore, when a countermodel is found the results are presented by running a familiar VHDL simulator. This makes the system more accessible to engineers. A weakness of the system is that manual intervention is sometimes required when the generated behavioural VHDL is not synthesisable, although this has rarely been the case in practice.

To illustrate the performance of LUCIFER we present the results of using it as a part of the Argus system to perform equivalence checking of various cores against their behavioural descriptions. Some of the results are shown in Table 1. These experiments were run on a dual processor Ultra-60 SparcStation with 2

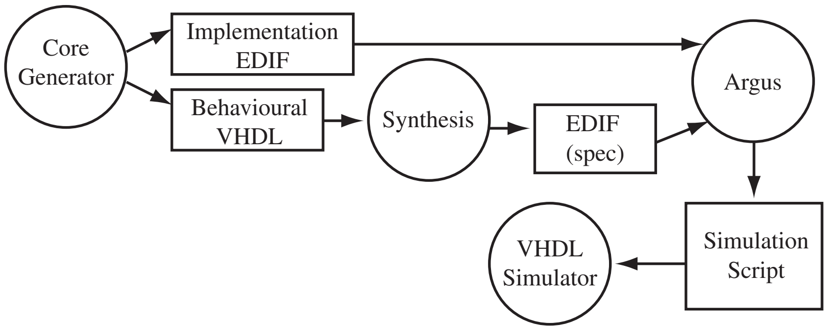

Fig. 4. Core Verification Flow at Xilinx

GB of RAM. Note that we verify *fixed size instances* of circuits, as you must expect of a system based on propositional logic.

Table 1. Experimental Results

Experiment	Verification Time (seconds)	Induction depth
64-bit up counter	1.03	0
64-bit up-down counter	75.39	0
11-bit loadable counter	2938	0
11-bit increment by 14 counter	0.16	0
11-bit increment by 14 loadable counter	2965	0
11-bit pipelined counter (1 stage)	46.21	2
11-bit pipelined counter (2 stages)	283	6
11-bit pipelined counter (3 stages)	526	10
1000-bit Johnson counter	44.45	2

The verification times for various n-bit up counters (not loadable, count by 1 only, no threshold outputs) are quite favorable. The netlist produced by the core generator system and behavioural synthesis system were examined. Both systems produced designs that contained D-type flip-flop state elements, but the core generator system used a D-type flip-flop with an enable signal whereas the synthesised version produced a D-type flip-flop active on a negative edge. These components are encoded with different logical representations so the system has to perform some non-trivial checking to ensure that the two counters are the same.

A counter which can be made to count up or down depending on a control signal has a more complicated state space, as suggested by the timing results. However, even for a 64-bit up-down counter, the verification takes little time (around one minute).

Keeping the count direction fixed but allowing the counter to be loaded with a new value on any clock tick proves to be more challenging for LUCIFER. It took about 50 minutes to verify an 11-bit loadable counter. Both the core generator and behavioural versions produce the same number of state elements but the core generator uses FDE flip-flops and the synthesiser used FD (D-type) flip-flops.

Verifying counters with specific increment values posed no problem for the verification with a 11-bit increment by 14 count taking just 0.16 seconds. The next experiment combines the up-down feature of the counter with the ability to dynamically load the counter. Once again the dynamic load requirement pushes the verification time to 2,965 seconds.

Next, we introduce a pipeline stage at the end of the counter. The verification is posed in such a way that the first output of the two circuits is ignored but then every subsequent output is required to be the same. Adding a pipeline stage causes a marked increase in verification effort. The time needed to verify a non-pipelined 11-bit adder is 0.17 seconds but adding one pipeline stage causes the verification time to shoot to 46.21 seconds. LUCIFER uses an induction depth of 2 to verify these counters. Adding another stage takes the verification time up to 283 seconds. We believe that this is due to the particular way the pipelining property has been expressed to LUCIFER and this is something we expect to be able to improve upon dramatically in the next iteration of verifications.

To study a larger example with unreachable states we verified Johnson counters at a variety of sizes (also known as twisted-ring counters). A Johnson counter counts in a sequence in which only one bit changes per clock tick. An n-bit Johnson counter cycles through $2n$ states, giving $2^{(n/2)} - n$ unreachable states. A 1,000 bit Johnson counter took 44.45 seconds to verify with induction depth 2. Verification of the ring counter presented earlier yields similarly encouraging results.

8 Discussion and Conclusion

We first presented a method of safety property checking based on induction with depth, strengthened with a constraint that all states in a path be unique. This method is complete. It is the method implemented in the prototype LUCIFER tool for analysis of Lustre programs [10]. The work on FPGA core verification described above is based on LUCIFER. Thus, these first results make use of strengthened induction with depth, and show that it can cope with non-trivial equivalence checking. The results also demonstrate that the method can be incorporated into a real design flow. The fact that erroneous behaviour found during attempted verification can be analysed in a standard VHDL simulator is particularly important. Our effort has so far been concentrated on the considerable task of building the infrastructure to automatically verify real cores in an existing design environment. The results reported have been produced only just in time for inclusion here, and we have not yet had time to analyse them. It is intended to continue this work by verifying a sequence of increasingly complicated cores. Those that are next on the list are large shifters and state machine based controllers. Thus, the development of both the methods and the infrastructure will be driven by real case studies. We expect to have to trim the safety property checking methods to match exactly this application to core verification. By doing so, we expect to reduce the verification times reported here considerably. We have a great deal of experimental work ahead, both in applying our methods and in comparing with more standard BDD-based verification methods.

We have also shown that the strongest form of induction is very close to a standard backwards and forwards analysis using fixpoints. The FixIt system, developed by Bjesse and Eén, implements many safety property checking algorithms, including BMC, strengthened induction with depth, and SAT-based versions of standard fixpointing algorithms. It makes use of a relatively simple quantifier eliminator, but still gives very promising results [1]. FixIt is now being used as a basis for experiments in SAT-based verification. We plan to use FixIt to investigate a variety of induction-based methods, including those presented in the previous section.

When using these induction-based methods, one can vary not only the induction strength, but also the transition relation, T. We can try to make the relation smaller (in ways that do not affect the final result of the analysis) so as to prune away paths, and so possibly reduce the necessary induction depth. The application of van Eijk's method by Bjesse and Claessen can be seen as an example of this [5]. Also, making the transition relation (viewed as a set of pairs of states) larger, while leaving the transitive closure unchanged, may cause the depth of induction needed to prove a property to be reduced. So, the challenge is to increase the size of the relation while leaving its transitive closure unchanged. It seems likely that we can use insights from relational algebra here. One can also think of eliminating some of the quantifiers in the unwindings of the transition relation (as distinct from in the constraints). This would give a sort of hybrid between the usual fixpoint methods and the inductive methods presented here. And there are many tricks from the world of BDD-based model checking that we haven't even considered yet! We have begun to think that a possible way to proceed might be to go back to basics and think of all these methods, and possible new ones, in terms of propositional temporal logic theorem proving. Insights from proof theory might then give us a new way to compare the different methods.

Acknowledgments

This research has been funded by Prover Technology AB, Xilinx, Inc., Chalmers University of Technology, the Swedish funding agency TFR, and the EU LTR project SYRF (Synchronous Reactive Formalisms). Carl Johan Lillieroth built major parts of the infrastructure for core verification. Many thanks to Per Bjesse, Koen Claessen and Gordon Pace for their constructive criticism of an earlier draft. Thanks also to Nicolas Halbwachs, Pascal Raymond and Bernie New for enlightening discussions.

References

1. P. A. Abdulla, P. Bjesse and N. Eén: Symbolic Reachability Analysis based on SAT solvers, In Proc. Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'00, LNCS, Springer-Verlag, 2000.
2. A. Biere, A. Cimatti, E.M. Clarke and Y. Zhu: Symbolic Model Checking without BDDs. In Proc. Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'99, number 1579, LNCS, Springer-Verlag, 1999.
3. A. Biere, A. Cimatti, E.M. Clarke, M. Fujita and Y. Zhu: Symbolic model checking using sat procedures instead of BDDs. Design Automation Conference, DAC'99, IEEE Press, 1999.
4. A. Biere, E.M. Clarke, R. Raimi and Y.Zhu: Verifying Safety Properties of a PowerPC Microprocessor Using Symbolic Model Checking without BDDs. In Proc. Int. Conf. on Computer-Aided Verification, CAV'99, LNCS, Springer-Verlag, 1999.
5. P. Bjesse, K. Claessen: SAT-based Verification without State Space Traversal. In Proc. Int. Conf. on Formal Methods in Computer Aided Design of Electronic Circuits, FMCAD'00, LNCS, Springer-Verlag, 2000.
6. E. Clarke, O. Grumberg and D. Peled: *Model Checking*, MIT Press, 1999.
7. W.J. Fokkink and P.R. Hollingshead: Verification of Interlockings: From Control Tables to Ladder Logic Diagrams, in (J.F. Groote, S.P. Luttik and J.J. van Wamel, eds) Proc. 3rd Workshop on Formal Methods for Industrial Critical Systems, FMICS'98, Amsterdam, 1998.
8. D. Deharbe and A. Martins Moreira: Using Induction and BDDs to Model Check Invariants, In H. Li and D. Probst, editors, Advances in Hardware Design and Verification, IFIP – Advanced Research Working Conference on Correct Hardware Design and Verification Methods: CHARME'97, Chapman and Hall, 1997.
9. C.J. Lillieroth and S. Singh: Formal Verification of FPGA Cores. Nordic Journal of Computing 6, 27-47, 1999.
10. M. Ljung: Formal Modelling and Automatic Verification of Lustre Programs Using NP-Tools, Master's thesis, Prover Technology AB and Department of Teleinformatics, KTH, Stockholm, 1999.
11. M. Sheeran and G. Stålmarck: A tutorial on Stålmarck's proof procedure for propositional logic. Formal Methods in System Design, 16:1, January 2000.
12. M. Sheeran and G. Stålmarck: Checking safety properties using induction and boolean satisfiability. Appendix to deliverable d20.2, EU project CRISYS, 1999.
13. G. Stålmarck: Stålmarck's Method and QBF Solving. In Proc. Int. Conf. on Computer-Aided Verification, CAV'99, LNCS, Springer-Verlag, 1999.
14. Xilinx: Xilinx IP Center, `http://www.xilinx.com/ipcenter`.