Verifying Properties Using Sequential ATPG

Jacob A. Abraham and Vivekananda M. Vedula
Computer Engineering Research Center
The University of Texas at Austin
Austin, TX 78712
{jaa, vivek}@cerc.utexas.edu

Daniel G. Saab
Department of Electrical Engineering and Computer Science
Case Western Reserve University
Cleveland, Ohio, 44119
saab@eecs.cwru.edu

Abstract

This paper develops a novel approach for formally verifying both safety and liveness properties of designs using sequential ATPG tools. The properties are automatically mapped into a monitor circuit with a target fault so that finding a test for the fault corresponds to formally establishing the property. The mapping of the properties to the monitor circuit is described in detail and the process is shown to be sound and complete. Experimental results show that the ATPG-based approach performs better than existing verification techniques, especially for large designs.

1. Introduction

As integrated circuit designs continue to increase in size, validating their correctness is becoming more and more difficult. It has been estimated that over 50% of the design effort is now in verification, and this percentage is expected to worsen in the future. Simulation of the design using validation tests is the main technique used in industry to validate large designs. Unfortunately, as designs become larger, the likelihood that the tests will uncover subtle bugs becomes smaller. Although simulation is expected to remain a key technique to validating complex designs, there is interest in the use of formal techniques as a complementary approach.

Unlike simulation which attempts to discover the presence of bugs in a large design, formal techniques are geared toward establishing the correctness of parts of the design. Since the entire design space has to be searched (at least implicitly) in order to establish definite correctness, formal approaches are only applicable to small portions of a design. (A module of a design which has around 300 state elements has more possible states than the number of protons in the universe!)

1.1. Formal verification in the design process

Formal verification is, at present, primarily used in establishing the equivalence between the register-transfer level (RTL) and the logic level. The widely used industry tools require a correspondence between the state elements in the RTL description and the logic-level description, essentially dealing with a *combinational equivalence* problem. This is still a very difficult problem (establishing the equivalence of two Boolean functions is NP-complete), but the use of good heuristics and a combination of different techniques, including binary decision diagrams (BDDs), ATPG and satisfiability (SAT) solvers, allows industry tools to formally verify the Boolean equivalence of combinational blocks in real designs.

Verifying the equivalence between the RTL and logic level is sometimes called *implementation verification*, since both levels are described formally and one is a refinement of the other. Verifying the correctness of the RTL itself is a different matter, since it has to be verified against the behavioral level which is usually described using a natural language. In order to deal with the lack of a formal description, designers usually generate formal *properties* (discussed in more detail in the next section) that the design must hold from the natural language description. The RTL is then formally checked whether the properties hold, and the process is called *design verification*. (This is similar to the development of test benches for simulation.) The focus of this paper is on design verification, i.e., property checking.

It should be noted that the primary benefit of formal approaches in industry has been their help in uncovering bugs.

Unfortunately, existing tools for formal analysis of designs are limited in the size of circuits they can handle. The objective of this paper is to develop an approach that a designer can use to formally check properties of much larger modules, in a framework which is compatible with the existing design flow.

2. Checking Properties

This section will briefly describe the types of properties which can be checked formally and the techniques used by the formal verification community to apply them to designs.

2.1. Safety and Liveness properties

Properties which express system behavior can be classified as safety and liveness properties [1].

- A **safety property** expresses the fact that *something bad will never happen*.
- A **liveness property** expresses the behavior whereby *something good will eventually happen*.

These properties are generally specified using *temporal logic* and variations, following work in verifying reactive programs [2], since digital systems receive inputs and produce outputs in a continuous interaction with their environment. For instance, using this logic one can express the assertion that if proposition p holds in the present, then proposition q holds at some instant in the future.

In *Linear Time* temporal logic (LTL), the notion of time is that of a linearly ordered set (this can be thought of as a possible sequence of states). Four operators are used to describe LTL properties in dealing with hardware verification, **F**, **G**, **U** and **X**.

- $\mathbf{F}q$ is true in the present if at some moment in the future q is true.
- $\mathbf{G}q$ expresses the fact that q is true at every moment of the future.
- $p\mathbf{U}q$ means that q will hold true at some moment in the future until which time p will hold at all moments.
- $\mathbf{X}q$ is true in the present if q is true in the next instant of time.

The **U** operator is called a *weak until* if q does not necessarily hold in the future, and a *strong until* if q definitely holds in the future.

Another temporal logic framework, *Computational Tree Logic* (CTL) is used to express the fact that at each instant of time there exist many possible futures [3]. Each branch is defined as a maximal linearly ordered set of states. Truth or falsehood of tense formulas are thought of as being relative to a given branch of the tree ordered frame.

- q is *necessarily* true is represented by the formula $\mathbf{A}q$ (i.e., along all branches).
- q is *possibly* true is represented by the formula $\mathbf{E}q$ (i.e., along some branch).

To illustrate the use of these operators to express safety and liveness properties, consider the design of a processor. It fetches instructions, decodes them, fetches operands, executes the instructions, writes the results back and returns to the fetch state. The design of the control for the processor includes the above sequence of states, and may also include illegal states (depending on the state assignment).

A *safety property* could be that the processor never enters an illegal state, say S_i. This can be expressed as $\mathbf{G}\neg S_i$ in LTL. If we wish to find a counterexample (a bug), we can check for the negative of the property, expressed in CTL as $\mathbf{EF}S_i$, i.e., along some execution path, at some time in the future, the processor enters an illegal state.

A *liveness property* could be that the processor never hangs, i.e., it will always return to the fetch state, S_f. This can be expressed in LTL as $\mathbf{F}S_f$. A counterexample for this would be an infinite sequence of states which does not include the fetch state, expressed in CTL as $\mathbf{EG}\neg S_f$, i.e., along some execution path, for all future, the fetch state is not reached.

2.2. Model checking

The process of analyzing a design for the validity of properties stated in temporal logic is called model checking. The input to a model checker is a formal description of the design, and the result is a set of states which satisfy the given property, or a witness of a sequence which violates the property.

McMillan [4] developed efficient techniques to manipulate Boolean formulas in model checking using Ordered Binary Decision Diagrams (OBDDs). Clarke et al. [5] showed that LTL model checking can be reduced to CTL model checking with fairness constraints.

2.3. Bounded model checking

The use of OBDDs allow the analysis of designs without explicitly enumerating their states. However, OBDDs are also vulnerable to the state explosion problem for even moderately complex designs. In practice, designers know the bounds on the number of steps within which a property should hold. This leads to the idea of *bounded model checking* where the property is determined to hold within a finite sequence of state transitions. (This notion is very familiar in the test generation community as the number of time frames explored in finding a test.) Another development was the use of Satisfiability Solving (SAT) algorithms

instead of BDDs to address the problem. In the SAT approach, a set of propositional clauses is generated from the design under consideration and this is checked for satisfiability. Larrabee [6] proposed the use of SAT techniques for combinational ATPG, and Konuk [7] made an extension to sequential ATPG. The sequential ATPG using SAT techniques assumed that the circuits had a reset state, and the experiments only compared the smaller benchmark circuits to other academic ATPG tools.

Biere et al. [8] introduced a bounded model checking procedure for LTL properties, using SAT algorithms instead of BDDs. This paper is based on the fact that bounded model checking for LTL can be reduced to propositional satisfiability in polynomial time; the "universal" model checking problem of the type $\mathbf{F}(x = 0)$ can be translated into the "existential" model checking problem $\mathbf{EG}(x \neq 0)$ by negating the formula. Then, a check is made whether there is an execution sequence that fulfills $\mathbf{G}(x \neq 0)$, restricting the search to paths that have at most $k + 1$ states. The suggestion in the paper is to initially restrict the bound k, and progressively increase it, looking for longer and longer possible counterexamples. Copty et al. [9] described the benefits of model checking at Intel, on designs taken from the Pentium 4 processor. A BMC checker with a SAT solver is reported to be superior to one with a BDD package.

3. Previous work in using ATPG for property checking

A counterexample to a safety property is a case where something bad happens in the circuit. Using our previous example, the bug (the "bad" event) is the processor going into an illegal state. An ATPG tool could be used to find the counterexample, by justifying the illegal state to a starting state (or the X-state). There has been some work in this direction.

Keller [10] suggested the use of ATPG engines to perform an examination of search spaces, but did not mention the application to verification. The use of sequential ATPG for model checking was proposed by Boppana [11]. This work focused only on safety properties, and studied the efficiency of sequential ATPG algorithms for state-space exploration. They noted that the main benefit of ATPG was the fact that there is no explicit storage of states at each time-frame, and its strength is a balance between a purely breadth-first search (used by model checkers), and purely depth-first search (inefficient for exploring large state spaces).

Cheng [12] discussed the use of ATPG for verification, including combinational equivalence checking and "property checking". The properties suggested included tristate bus contention and asynchronous feedback loops. The solution was to map the property to a combinational circuit and test for a stuck-at fault at the output of the circuit, essentially checking a safety property (Fp, that p is true some time in the future).

ATPG and SAT algorithms were compared by Parthasarathy [13]. This work identified some of the tradeoffs between the two, including the fact that ATPG could deal naturally with real-world primitives such as tri-state buses and high-impedance logic values. Their results on combinational circuits from industry and static circuit properties showed that there was no performance gap between SAT and ATPG solvers. Our results on benchmark circuits for temporal logic properties shows a clear advantage for ATPG techniques for small circuits, with model checking based on SAT unable to deal with the biggest designs.

Huan [14] proposed a combination of structural, word-level sequential ATPG and modular arithmetic constraint-solving techniques for checking safety properties such as bus contention checking, internal don't-care validation and invariant checking. These were transformed into a counterexample generation problem to be solved by a custom ATPG engine.

Hsiao [15] also suggested the use of sequential ATPG for verifying safety properties of the type EFp and compared this with OBDD-based approaches using a simulation-based ATPG (with a genetic algorithm). They also noted that ATPG could perform the state-space search without needing the complete state-space information at one time. They found that, although only justification was needed to check the safety properties using ATPG, the incomplete but useful information learned via propagation can improve the performance of ATPG for the property checking.

Although checking for a safety property (more precisely, finding a counterexample to a safety property) can be easily done using ATPG techniques, it is not clear how liveness properties can be checked. A liveness property can be checked directly (examining all states to ensure that the property holds), or we can look for a counterexample (a sequence of states which contradicts the property). Using the processor example above, a counterexample to the property that the processor will eventually go to the start state is a sequence of states which does not include the start state. It is not obvious how ATPG can be used to find such a sequence. We develop a solution below.

4. Our approach

A key requirement for a technique to be usable by designers is that it fit seamlessly within the normal design flow. We use any existing ATPG tool, without modifications, with a circuit description which is compatible with the tool, and add a small circuit which will guide the ATPG tool into checking for a desired property.

We will consider properties stated as temporal logic formulas of the type $\mathbf{A}p$, where p is a *restricted path formula*

in which the only state sub-formulas are atomic propositions [5]. Our approach automatically maps both safety and liveness properties as well as a bound on the number of steps to be verified into a *monitor circuit with a target fault*, so that any existing ATPG tool can be used to test for the fault. A test for the fault becomes a witness for the property. If the fault is determined to be untestable, the property is guaranteed to hold within the bound. If ATPG aborts, we cannot say anything about the property.

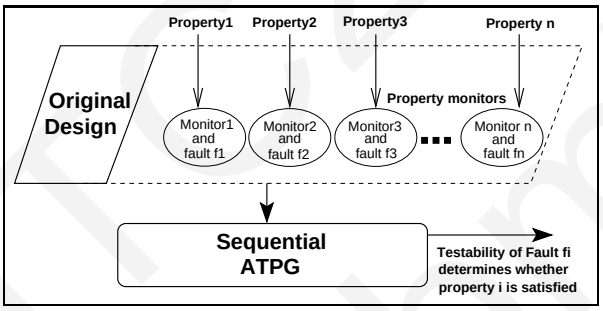

Figure 1. Approach to property verification using ATPG

Figure 1 illustrates the essential features of our approach. We generate property monitors for the properties of interest (these use the relevant signal names from the original design), but the original design is not modified. The design and the monitors with the target faults are given to any sequential ATPG tool, and the result of the test generation for the faults determine whether the corresponding property is satisfied.

The advantage of such an approach is that we can deal with any design without direct modifications to the circuit implementing it (we use an external monitor circuit which just observes circuit nodes to guide the ATPG). We are also able to exploit the power of existing ATPG tools. Other approaches to using ATPG for validation require the use of a custom (non-commercial) ATPG [15] as well as modifications to the ATPG tool [14]. Given the extreme complexity of commercial ATPG tools, very few organizations will be able to make modifications to them.

Each of the steps in the refinement of the design involves heuristics and optimizations, so a verification at one stage in the design process does not mean that the next stage of the design will remain correct. Another advantage of using ATPG is that the verification is performed on a model which is much closer to the final hardware (at the logic gate level). In addition, the fact that the ATPG tool is used to generate the final tests on the hardware will give a high degree of credibility to ATPG-based verification.

5. Mapping properties to stuck faults

This section describes the mapping of the formal properties into monitor circuits with target faults. Many organizations use monitors to check the outputs of simulation, so this concept is familiar to designers. We first describe the framework in which ATPG will be used, then provide details of the monitors and establish their validity.

5.1. Framework for property checking using ATPG

Since ATPG is well suited to searching for a test (a counterexample), we will use the fact that "universal" properties can be reduced to "existential" properties [8]. Thus, in order to test for the LTL property with a bound of n, $\mathbf{F}p$ (at some time in the future within n cycles, p will be true), we use ATPG to find a sequence of length n where p is not true (i.e., we attempt to find a witness for $\mathbf{EG}\neg p$. The finite-state machines representing the monitor circuits for each of the properties, $\mathbf{EF}p$, $\mathbf{EG}p$, $\mathbf{X}p$ and $p\mathbf{U}q$ [1], are described below for a bound of n. If the final (absorbing) state is reached, the property will be satisfied. The fault to be tested for would simply be the output of a gate which decodes this final state.

ATPG algorithms generally assume that the circuit starts in an unknown state (all "Xs"), then finds a reset or synchronization sequence to a known state. In the checking framework, we start the circuit in the all "X" state and the monitor in its starting state. If a test is found for the target fault corresponding to a property, there will be a sequence, starting in the all-X state, which is a witness to the property. The state (or set of states) which is within n cycles from the final state satisfies the property. As we will see in the results, this framework is consistent with the one used in classical model checking, which tries to find the set of states which satisfy a property. If ATPG determines the target fault corresponding to a property to be untestable, then it is guaranteed that there is no sequence of states of length $\leq n$ which will satisfy the property. Finally, if ATPG aborts, we cannot say anything about the property.

5.2. Monitors for existential properties

Figure 2 shows the monitor for the $\mathbf{EF}p$ property. The search starts in the state indicated (state 1), and if at any time p is satisfied, we go to the final state, n. The target fault for the ATPG would be a stuck fault on the output of a gate decoding the state n. If ATPG finds a test for the fault, we will know that the property is satisfied, and if the fault is untestable, then p will not be true for any sequence sequence of length $\leq n$.

Note that if we did not need to include a bound (n), then $\mathbf{EF}p$ could be checked by creating a combinational subcircuit which is true if p is true, and asking ATPG to generate a test for a stuck-at-0 at the output of this circuit. This is the approach taken by previous solutions to checking for

[1] Patent applications are being submitted for these circuits and for the process of using them to verify properties with an ATPG engine

safety properties. Our state machine incorporates the bound and is consistent with the state machines for the liveness properties.

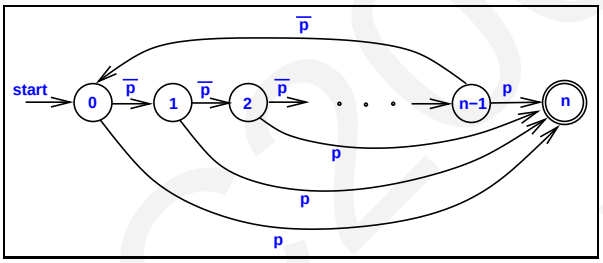

Figure 2. Monitor for $\mathbf{EF}p$

Figure 3 shows the monitor for the property $\mathbf{EG}p$ for a bound of n. Let us look at this finite-state machine (FSM) in more detail, since it was not known previously how such a liveness property could be checked using ATPG. In this FSM, we progress from the start state for n steps to the final state labeled n, only if p is true at each step. If at any step, we are in a state where p is not true, we go back to the starting state.

In order to see why this monitor will force the ATPG to generate a witness for $\mathbf{EG}p$, let us look at the cases where ATPG finds a test for a fault on the gate decoding the final state (labeled n) and where it cannot find a test.

Suppose ATPG finds a test for the fault. Then the test will include a sequence of n states where at each state p is true, which is indeed a witness for $\mathbf{EG}p$. If the test from the all-"X" state is longer than n steps, the last n steps would be the witness sequence, and the set of states at the beginning of the sequence satisfy $\mathbf{EG}p$.

Suppose ATPG determines that the fault is untestable. In that case, it has determined that there is no sequence of inputs which will produce a sequence of n steps where p holds at each step of the sequence, indicating that the property is not satisfiable in n steps.

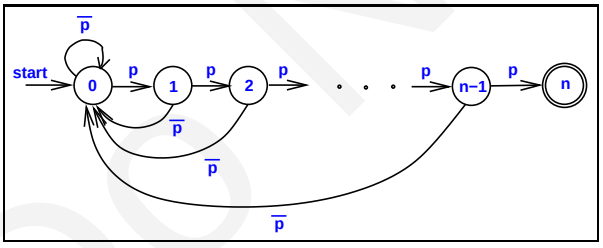

Figure 3. Monitor for $\mathbf{EG}p$

Figure 3 is the monitor for the property $\mathbf{EX}p$. It is quite easy to see that the test produced by the ATPG will indicate a state where p is true in the next state.

Finally, figure 5 shows the FSM for the $\mathbf{E}p\mathbf{U}q$ property. This implements the "strong until" which would be more of interest to circuit designers. The argument for why this circuit would guide ATPG into finding a witness for the property is similar to the one for the $\mathbf{EG}p$ property above.

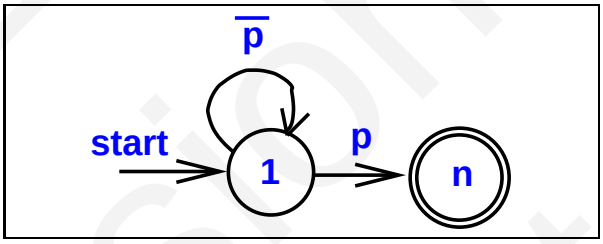

Figure 4. Monitor for $\mathbf{EX}p$

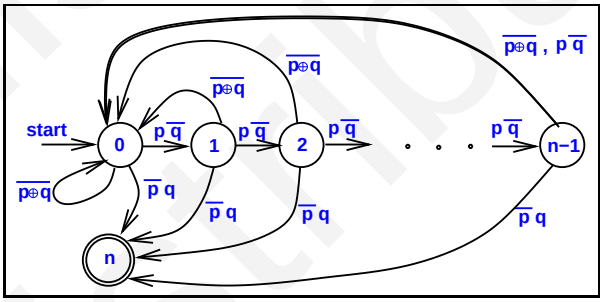

Figure 5. Monitor for $\mathbf{E}p\mathbf{U}q$

It is interesting that the two non-trivial FSMs are for the liveness properties, and represent the major contribution of this paper.

6. Results

We now provide experimental results of applying the novel technique. We automatically mapped the existential properties into the monitor circuit which was attached to the design. A commercial sequential ATPG tool [16] was used to generate the witnesses. The results were compared with a free research version of a bounded model checker (originally from Carnegie-Mellon University) from Cadence Berkeley Labs [17]. We generated a set of simple properties on the sequential ISCAS 89 benchmark circuits for the comparison of the two approaches.

All experiments were performed on a Sun Microsystems Ultrasparc II system with dual processors running at 450 MHz, and with 1 GByte of memory.

6.1. Safety properties

For each of the benchmark circuits, we chose the LTL property $\mathbf{G}((o_1 \cdot o_1 \cdot o_2 \cdot \ldots \cdot o_k) = 0)$, where o_i is the i th output of the circuit, i.e., we wanted to check if for all times in the future the circuit would have at least one output which is zero. ATPG is used to find a witness for $\mathbf{EF}((o_1 \cdot o_1 \cdot o_2 \cdot \ldots \cdot o_k) = 1)$, i.e., it tries to find a state where all outputs are 1 simultaneously. Table 1 gives the results of this experiment. The first four columns give the

Table 1. EF property for AND of all outputs (Bound = 15)

Circuit Name	Primary In + Out	Comb. Gates	Seq. Elem.	EF_AND_1		
				Counter-Example?	ATPG CPU(s)	BMC CPU(s)
s27	6+1	10	3	Yes	0.1	0.22
s820	20+19	289	5	No	0.1	1.52
s832	20+19	287	5	No	0.1	1.49
s1488	10+19	653	6	No	0.2	2.61
s1494	10+19	647	6	No	0.1	2.73
s386	9+7	159	6	No	0.1	0.72
s510	21+7	211	6	No	0.1	0.82
s208.1	12+1	104	8	Yes	0.1	0.73
s298	5+6	119	14	No	0.2	0.96
s344	11+11	169	15	No	0.1	3.32
s349	11+11	170	15	No	0.1	4.64
s420.1	20+1	218	16	Yes	0.1	1.53
s1196	16+14	529	18	No	0.1	3.17
s1238	16+14	508	18	No	0.1	3.27
s641	37+24	380	19	No	0.1	2.68
s713	37+23	393	19	No	0.1	2.43
s382	5+6	158	21	No	0.2	1.25
s400	5+6	164	21	No	0.1	1.26
s444	5+6	181	21	No	0.1	1.44
s526	5+6	193	21	No	0.2	1.68
s953	18+23	395	29	No	0.2	3.4
s838.1	36+1	446	32	Yes	0.1	3.48
s1423	19+5	657	74	Yes	2.2	5.15
s5378	37+49	2779	179	No	0.3	10.22
s9234.1	38+39	5597	211	No	0.3	24.5
s9234	21+22	5597	228	No	0.3	24.29
s15850.1	79+150	9772	534	No	1.1	78.23
s15850	16+87	9772	597	No	0.9	76.85
s13207.1	64+152	7951	638	No	0.7	65.63
s13207	33+121	7951	669	No	0.7	64.14
s38584.1	40+304	19253	1426	No	3.6	-
s38584	14+278	19253	1452	No	2.9	-
s38417	30+106	22179	1636	No	28	-

circuit name and its details (number of I/O pins, combinational gates and sequential elements). The next column indicates whether or not a counterexample exists; the property is satisfied for some circuits but not for others. The last two columns show the CPU times in seconds for ATPG and BMC, respectively. BMC was unable to generate the SAT clauses for the largest circuits (indicated by a "-" in the table).

6.2. Liveness properties

The liveness property we devised for the benchmark circuits was $\mathbf{F}((o_1 \cdot o_1 \cdot o_2 \cdot \ldots \cdot o_k) = 0)$, where o_i is the i output; i.e., there is a sequence of states of length n, the bound chosen, within which some output will be zero. ATPG tries to find a witness to the counterexample, $\mathbf{EG}((o_1 \cdot o_1 \cdot o_2 \cdot \ldots \cdot o_k) = 1)$, i.e., a sequence of states of length n for which all outputs are simultaneously 1. It does this by attempting to generating a test for s-a-0 on the property monitor circuit, which ensures that $p = o_1 \cdot o_1 \cdot o_2 \cdot \ldots \cdot o_k = 1$ at every state along the path.

The results are presented in Table 2 for different values of the bound, n, from 5 through 25. For each bound, the first column indicates whether a counterexample to the property exists, and the next two columns give the times taken by ATPG and BMC, respectively. Again, BMC was unable to deal with the largest circuits.

Table 2. EG_AND property for various bounds

Circuit Name	Bound = 5			Bound = 10			Bound = 15			Bound = 20			Bound = 25		
	Ctr.-Ex.	ATPG sec.	BMC sec.	Ctr.-Ex.	ATPG sec.	BMC sec.	Ctr.-Ex.	ATPG sec.	BMC sec.	Ctr.-Ex.	ATPG sec.	BMC sec.	Ctr.-Ex.	ATPG sec.	BMC sec.
s27	Yes	0.1	0.21	Yes	0.1	0.19	No	0.2	0.21	No	0.2	0.25	No	0.2	0.31
s820	No	0.1	0.91	No	0.2	1.05	No	0.1	1.21	No	0.1	1.45	No	0.2	1.75
s832	No	0.1	0.92	No	0.2	1.06	No	0.1	1.34	No	0.1	1.44	No	0.1	1.73
s1488	No	0.1	1.54	No	0.2	1.79	No	0.2	2.02	No	0.1	2.38	No	0.2	2.95
s1494	No	0.2	1.51	No	0.2	1.85	No	0.1	2.05	No	0.1	2.37	No	0.2	3.1
s386	No	0.1	0.46	No	0.2	0.53	No	0.1	0.58	No	0.1	0.69	No	0.1	0.88
s510	No	0.2	0.54	No	0.1	0.65	No	0.2	0.8	No	0.2	0.88	No	0.1	1.09
s208.1	Yes	0.1	0.4	Yes	0.01	0.59	No	0.3	0.65	No	0.2	0.86	No	0.3	1.08
s298	No	0.5	0.42	No	0.4	0.49	No	0.4	0.59	No	0.4	0.79	No	0.4	0.93
s344	No	0.1	0.86	No	0.1	1.27	No	0.1	1.73	No	0.1	2.2	No	0.2	4.73
s349	No	0.1	0.89	No	0.1	1.29	No	0.1	1.91	No	0.1	2.48	No	0.1	4.39
s420.1	Yes	0.2	0.71	Yes	0.1	1.09	No	0.3	1.47	No	0.3	1.85	No	0.4	2.73
s1196	No	0.1	1.2	No	0.1	1.43	No	0.1	1.73	No	0.1	1.91	No	0.01	2.35
s1238	No	0.2	1.24	No	0.2	1.51	No	0.1	1.74	No	0.1	1.99	No	0.1	2.54
s641	No	0.1	0.91	No	0.1	1.07	No	0.1	1.24	No	0.1	1.56	No	0.1	1.86
s713	No	0.2	0.94	No	0.1	1.08	No	0.1	1.19	No	0.1	1.36	No	0.1	1.57
s382	No	0.2	0.56	No	0.1	0.55	No	0.1	0.68	No	0.2	0.77	No	0.1	0.98
s400	No	0.1	0.52	No	0.1	0.59	No	0.1	0.71	No	0.1	0.86	No	0.1	0.98
s444	No	0.2	0.51	No	0.1	0.64	No	0.1	0.69	No	0.1	0.82	No	0.1	1.01
s526	No	3.6	0.64	No	3.3	3.65	No	4.2	3.94	No	3.6	40.45	No	3.5	168.6
s953	No	0.1	1.16	No	0.1	1.32	No	0.2	1.68	No	0.2	1.83	No	0.2	2.16
s838.1	Yes	0.1	1.57	Yes	0.2	2	No	0.3	2.88	No	0.4	3.77	No	0.5	5.28
s1423	No	2.2	1.75	No	2.1	2.18	No	2.3	2.63	No	2.2	3.2	No	2.2	3.65
s5378	No	0.2	7.13	No	0.3	7.75	No	0.3	8.77	No	0.2	9.46	No	0.3	11.75
s9234.1	No	0.5	10.83	No	0.6	11.2	No	0.5	12.53	No	0.5	13.67	No	0.5	14.95
s9234	No	0.4	10.49	No	0.5	11.66	No	0.5	12.34	No	0.4	13.23	No	0.5	14.82
s15850.1	No	0.9	32.08	No	1	35.87	No	1	36.77	No	1	40.77	No	0.9	45.55
s15850	No	1	32.81	No	0.9	35.51	No	0.799	36.87	No	0.9	41.19	No	1	46.41
s13207.1	No	0.6	25.99	No	0.7	28.95	No	0.6	30.95	No	0.6	33.67	No	0.7	38.68
s13207	No	0.6	25.67	No	0.6	28.45	No	0.699	30.73	No	0.6	33.37	No	0.5	38.6
s38584.1	No	3.4	-	No	3.4	-	No	3.3	-	No	3.3	-	No	3.2	-
s38584	No	2.8	-	No	3.1	-	No	2.9	-	No	2.9	-	No	2.7	-
s38417	No	3	-	No	2.8	-	No	2.9	-	No	2.8	-	No	2.8	-

6.3. Memory usage

The peak memory usage for BMC (SMV + the SAT solver (Zchaff)) is around 150MB for the largest circuit that it could handle (s13207). The program could not handle the larger designs. On the contrary, the peak memory requirement for ATPG (including the largest design) is only about 50 MB. However, for the smallest design (s27) ATPG uses about 34 MB, while BMC takes only around 1 MB. Though the SAT-based approach requires less memory for very small circuits, the memory requirement grows very quickly since the number of clauses generated grows with the size of the circuit and the bound, n. On the other hand, the memory requirement for ATPG grows very slowly with increasing size of design, since it is needed primarily for the representation of the circuit and the values of the nodes.

6.4. Selecting bounds

Table 2 also gives some interesting trends in the behavior of the two approaches as the bound, n, is increased. As mentioned earlier, the recommended approach when using bounded model checking is to start with a small bound, looking for a counterexample, and to progressively increase the bound to check whether the property holds for larger and larger bounds. The time required by BMC does increase as the bound is increased. In particular, for circuit s526, the time increases from less than a second for $n = 5$ to over 150 seconds for $n = 25$. In contrast, the time required by ATPG stays about the same, independent of the bound for this circuit. In real designs with large sequential depths, it is not always easy to guess an accurate value for the bound. An ATPG-based approach can start with a relatively large

bound for n without incurring much penalty in CPU time.

7. Conclusions

We have presented an approach for formal verification which fits within the normal design flow, and which does not require any modifications to the design to be verified. To our knowledge, this is the first time that the important class of *liveness* properties have been checked using ATPG techniques.

Our results show that ATPG-based search techniques are superior to conventional SAT-based search even for small problems, and that the ATPG-based approach is able to deal with much larger designs than is possible with SAT-based approaches. The ATPG-based approach is able to prove properties in designs with "real" artifacts such as tri-state buffers, and can be inserted seamlessly into the normal design flow, making it easy to apply to real designs. Finally, we believe that our techniques will help verify much more complex circuits in practice, and hope that more attention will be focused on this new and broader application for sequential ATPG.

We are continuing to generate results for more complex designs, including processors, and for other properties, including $p\mathbf{U}q$. These will be included in the final version of the paper.

References

[1] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Transactions on Software Engineering*, Vol. SE-3, no. 2, March 1977, pp. 124-143.

[2] A. Pnueli, "The Temporal Semantics of Concurrent Programs," *18th Symposium on Foundations of Computer Science*, 1977.

[3] E. Clarke, E. A. Emerson and A. Sistla, "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications," *ACM Trans. Programming Languages and Systems*, Vol. 1, no. 2, 1986, pp. 244–263.

[4] K. L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[5] E. M. Clarke, O. Grumberg and H. Hamaguchi, "Another look at LTL model checking", *Formal Methods in System Design*, Vol. 10, no. 1, February 1997, pp. 57–71.

[6] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiability," *IEEE Transactions on Computer-Aided Design*, January 1992, pp. 4–15.

[7] H. Konuk and T. Larrabee, "Explorations of Sequential ATPG Using Boolean Satisfiability," *Proceedings of the 11th IEEE VLSI Test Symposium*, 1993, pp. 85–90.

[8] A. Biere, A. Cimatti, E. M. Clarke and Y. Zhu, "Symbolic Model Checking without BDDs." *Tools and Algorithms for the Analysis and Construction of Systems, (TACAS'99), LNCS 1579*, March 1999, pp. 193–207.

[9] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella and M. Y. Vardi, "Benefits of Bounded Model Checking at an Industrial Setting," *Computer-Aided Verification (CAV01)*, July 2001, pp. 436–453.

[10] B. Keller, K. McCauley, J. Swenton and J. Youngs, "ATPG in Practical and non-Traditional Applications," *Proceedings of the International Test Conference*, October 1998, pp. 632–640.

[11] V. Boppana, S. P. Rajan, K. Takayama and M. Fujita, "Model Checking Based on Sequential ATPG," *Computer-Aided Verification*, July 99, pp. 418–429.

[12] K. T. Cheng and A. Kristic, "Current Directions in Automatic Test-Generation," *IEEE Design and Test*, Vol. 32, November 1999, pp. 58–64.

[13] G. Parthasarathy, C.-Y. Huang and K.-T. Cheng, "An analysis of ATPG and SAT algorithms for formal verification," *Proceedings High-Level Design Validation and Test Workshop*, November 2001, pp. 177–182.

[14] C.-Y. Huan and K.-T. Cheng, "Using word-level ATPG and modular arithmetic constraint-solving techniques for assertion property checking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 20, No. 3, March 2001, pp. 381–391.

[15] M. Hsiao and J. Jain, "Practical use of sequential ATPG for model checking: going the extra mile does pay off," *Proceedings Sixth IEEE International High-Level Design Validation and Test Workshop*, 2001, pp. 39–44.

[16] Mentor Graphics Corporation, http://www.mentor.com/flextest/

[17] Cadence Berkeley Laboratories, http://www-cad.eecs.berkeley.edu/ kenmcmil/smv/