Illegal State Space Identification for Sequential Circuit Test Generation*

M.H. Konijnenburg J.Th. van der Linden A.J. van de Goor
Faculty of Information Technology and Systems
Delft University of Technology, Delft, The Netherlands
E-mail: M.H.Konijnenburg@its.tudelft.nl

Abstract

Test generation for (synchronous) sequential circuits (STPG) has a huge search space, which often prevents successful test generation or untestability proof. In this paper four new techniques are proposed to expand the known Global Illegal State *(GIS) space, in order to reduce the search space. These techniques use the known GISes to generate candidate GISes, which have to be proven unjustifiable. This is an effective method to improve STPG performance because the number of stored GISes is reduced, saving memory and CPU time, while covering a larger part of the GIS space. To accelerate GIS space identification, we propose the* legal state cache*, to avoid useless justification repetitions. A data-structure is proposed to reduce the memory usage of the (G)ISes up to 10 times, and to accelerate GIS usage. Experimental results show a significant improvement in fault efficiency and CPU usage.*

1. Introduction

Many significant contributions to the field of STPG have been made [1...7], yet the fault efficiencies reached often are low and the required CPU times very large. Especially for larger circuits, STPG results are disappointing, illustrating STPG's complexity. A low density of *state-encoding* has been reported as the major cause of this high complexity[8]. Also in our experience, circuits generally have a low density of encoding; a large part of the state space is functionally not used and often the states in that part are unjustifiable. Such states have been named *Illegal States* (ISes) [1,4,7,8]. ISes often can be very sparsely assigned, i.e., they have only a few assigned entries compared to the total number of entries, causing the low encoding density. Consider a BCD-counter: it has 6 fully assigned ISes out of 16 states, which can be covered by 2 ISes: 1X1X and 11XX; for each BCD-counter, a circuit has two ISes with only 2 entries assigned.

*This work is partially funded by Philips Electronic Design and Tools.

This paper focuses on the identification of the *Global Illegal State* (GIS) space. A GIS is an unjustifiable state in the *fault-free* circuit, assuming an unknown initial state of the FlipFlops (FFs). Knowing an as large as possible part of the GIS space helps to constrain the STPG search space to justifiable states. As soon as a state inside the known GIS space is reached, STPG can backtrack. This avoids fruitless justification attempts, saving CPU time and increasing the fault efficiency.

(G)ISes are implicitly found during STPG when justification of some (fault-free) portion of the state fails[9][2][4]. However, the (G)ISes found in this way are rarely as sparsely assigned as they can be (i.e., they are *over-specified*), so that (too) many, less effective, (G)ISes are stored, preventing maximal profit, and possibly causing storage problems. Hence, efficient methods to find as effective as possible GISes, and to optimize a collection of GISes as it grows during STPG, are essential to STPG.

Prior art on illegal states in STPG includes: GIS *expansion* to remove over-specified entries of the known GISes, by simply unassigning assigned entries in a GIS one by one, and trying to justify the resulting Potential GIS (PGIS), is proposed in DUST[3]. Justification failure of PGISes results in a reduction of the number of GISes while the known GIS space is increased. This form of GIS expansion is also used in DAT[4]. In addition, two GIS optimization techniques were proposed which try to generate PGISes by *merging* GISes, based on pairs or sets of similar GISes.

Methods to obtain more effective GISes prior to STPG, have also been proposed. In [7], a process has been proposed to find GISes, using implicit state enumeration based on BDDs. Due to the use of BDDs the method is restricted to small circuits, or partitions. In [6], three techniques are proposed, using logic simulation of the *valid* states forward in time, to find the GISes. The major drawback is that all reachable states have to be found, in order to find any GISes. For larger circuits this will be less efficient. In [4], we proposed a simple pre-process *GIS learning* (called FIS in [4]) to find GISes. PGISes are generated with 1, 2, etc., assigned entries and tried to be justified. The identified GISes are

a) FF → 1 2 3 4 5 6
GIS
A 1 1 X X X X group 1
B X X X 1 0 X group 2
b) C X X X X X 0 new group formed
D 1 X 1 X X X extend group 1
E 0 X 0 1 X X merge group 1 and 2

FF 1,2 in group 1;GIS A associated to group 1
FF 4,5 in group 2;GIS B associated to group 2

Figure 1. FF-groups based on independent GISes; FF-groups are added or extended due to new GISes.

very effective because usually they are sparsely assigned.

In this paper we propose 4 techniques that improve or add to the 3 techniques from [4], to find new GISes and/or remove overspecification in existing GISes. It appeared that the effectiveness of the techniques in [4] is highly circuit dependent. We also noticed that storage and fast handling is an issue with (G)ISes. The number of GISes can be very high, urging for compression to limit memory usage. But this should not significantly decrease the access-time and comparison speed of GISes. Therefore, we propose the concepts of *FF-groups* and *compressed* vectors to improve storage and handling of (G)ISes. In order to speed-up GIS discovery, the *Legal State cache* (LS cache) is proposed to avoid useless repetitive justification of known legal states.

The remainder of this paper is organized as follows: Sections 2 and 3 describe the concepts of FF-groups and compressed vectors. Section 4 describes the proposed techniques to expand the known GIS space. Section 5 describes the LS cache. Experimental results are given in Section 6; while Section 7 summarizes the presented ideas.

2. Groups of flipflops

An *FF-group* is a group of FFs. Each GIS is associated to one FF-group. The GISes of one FF-group are dependent of each other in the sense that they have common assigned entries. The concept of FF-groups is based on the fact that GISes, in particular the sparse GISes, often do not have common assigned entries. Figure 1 shows five GISes $A \dots E$. When GIS A is stored in the GIS list, FF-group 1 is defined, consisting of the FFs of the assigned part of GIS A: FF 1 and 2. Storage of GIS B results in FF-group 2 (FF 4 and FF 5), because GIS B is independent of GIS A (the entries in GIS B corresponding to the FFs in group 1 are all X). During STPG runs, new GISes are stored in the GIS list, possibly resulting in changes in the FF-groups (see Figure 1b): New FF-groups can be formed (e.g., due to GIS C), FF-groups can be extended with FF(s) (e.g., insertion of GIS D in the list results in the extention of FF-group 1 with FF 3), or two or more FF-groups can be merged, because a new GIS is stored, that covers two or more FF-groups (e.g., GIS E). The use of FF-groups saves CPU time in two ways: First, if during STPG an FF is assigned a value, only the GISes which are associated to the FF-group of that FF have to be compared. E.g., if FF 1 is assigned a value, only GIS A has to be checked (see Figure 1a). Second, within a group, only the FFs of that group have to be compared to the corresponding entries of the GISes associated to that group. E.g., consider the two FF-groups in Figure 1a: Only assignments on FF 1 and 2 have to be compared in group 1; assignments on FF 4 and 5 in group 2.

3. Compressed vectors

A compressed vector (cvector) is a vector of values in which each value (the state of one FF) consists of three (assuming fault independent GISes) or six (fault dependent) bits, instead of an integer (usually 32 bits) in the case of a regular vector. Each bit in such a value corresponds to one of the basic values 0, 1 and U. In this way each value of a cvector is a *set* of basic values. Within one integer, ten 3-bit values (fault independent) or five 6-bit values (fault dependent) can be stored, resulting in compression rates of 10 or 5 respectively. An advantage of using set-based values is that, for example, states 0 X and 1 X are merged into a single state $0 \vee 1$X, resulting in further reduced storage requirements.

Cvectors reduce the storage requirements of ISes, while in addition the comparison speed between ISes and between the current PPI/PPO state and ISes is improved: The values of the cvectors are compared integer-wise, so 10 or 5 values are compared in one integer-level step.

4. GIS space discovery

In this section techniques are proposed to expand the known GIS space by GIS identification, expansion and merging. Based on the current GIS list, PGISes are generated which after justification failure become GISes. This has two advantages: the new GISes each cover one or more GISes in the list, which can be removed; and always a larger known GIS space is obtained. In [4], we proposed three techniques to expand the known GIS space:

T1: Expansion of GISes by generating PGISes for each assigned entry of a GIS, by unassigning that entry. Process Expand-Illegal-States (EIS, called MIS in [4]) executes technique T1 on the stored GISes, and is performed between two ATPG stages.

T2: PGIS generation from two similar GISes. This technique generates a PGIS from the common assigned fraction of two GISes, with the different entries set to X. Process

Check-Illegal-States (CIS) additionally applies technique T2, while searching for redundant (covered or duplicate) GISes.

T3: PGIS generation from ranges of similar GISes. This technique generates a PGIS based on the common assigned fraction of a *range* of *successively* stored GISes.

Although these techniques perform well for a large class of circuits, for many other circuits they are not or less effective, resulting in large GIS lists with many overspecified GISes. Therefore, additional techniques are needed to better handle these hard circuits. In the next four paragraphs, four such new techniques are proposed.

T4: Direct expansion of a new GIS. Often GISes are stored in the list, which would not be stored when earlier found GISes had been expanded using T1 [4], because then the new GISes would have been covered by these expanded GISes. However, process EIS, which applies technique T1 is performed between ATPG stages and not immediately after new GISes are stored.

Technique T4 solves this disadvantage of process EIS: Using a low backtrack limit (16 or 32), each new GIS that has at least a predefined minimum of assigned entries, is expanded by T1 *immediately* after this GIS is stored.

T5: PGIS from the common assigned fraction of the range of GISes identified during STPG for one fault. This technique improves and replaces technique T3. Experiments showed that it was very difficult to make T3 successful for a circuit, because the technique (due to parameter *weight* [4]) is extremely fault and circuit dependent. T5 modifies T3 in such way that it becomes highly independent of circuit and fault. Technique T5, like T3 [4], detects common assigned fractions in ranges of successively stored GISes. The common fraction has been decided/implied by the STPG to detect the fault-under-test, but is found to be unjustifiable later on in the STPG-process. However, after the common fraction was implied, (and before that fraction is being justified), often more decisions are implied (on the PPIs) to test the fault. These later decisions are not the reason this state is unjustifiable, resulting in (successively stored) over-specified GISes (with a common fraction).

Figure 2a shows five GISes which have been determined during STPG for a fault. The common fraction of these five states (X1X01XX) is probably the reason that these five states have been declared illegal. Figure 2b shows how the common fraction is detected. An array of integers, called *common_fraction_array* (cf), is updated each time a GIS is stored in the list, by comparing each entry i of the GIS ($GIS[i]$) to each entry i of the previously added GIS ($prev.GIS[i]$) according to the following three rules:

Rule 1. if $GIS[i] = X$ then $cf[i] = 0$

Rule 2. if $GIS[i] = b \wedge prev.GIS[i] = b$ ($b \in \{0,1\}$) then $cf[i] = cf[i] + 1$;

a) PPI-> 1 2 3 4 5 6 7
ISes in GIS list:

	1	2	3	4	5	6	7
1	X	**1**	1	**0**	**1**	0	0
2	1	**1**	X	**0**	**1**	0	0
3	0	**1**	X	**0**	**1**	1	1
4	X	**1**	1	**0**	**1**	0	1
5	X	**1**	0	**0**	**1**	1	0

b) common_fraction array (cf)

0	1	1	1	1	1	1
1	2	0	2	2	2	2
1	3	0	3	3	1	1
0	4	1	4	4	1	2
0	5	1	5	5	1	1

common fraction == Potential GIS: **X 1 X 0 1 X X**

Figure 2. PGIS from the common assigned fraction of a range of GISes.

GIS \ FF	1	2	3	4	5	6	
A	1	1	X	X	X	X	FF-group 1
B	X	X	X	1	0	X	FF-group 2
C	0	X	0	1	X	1	merge group 1 and 2
PGIS 1	0	X	0	X	X	X	to avoid merging
PGIS 2	X	X	X	1	X	1	

Figure 3. Generate two PGISes to avoid merging of two FF-groups.

Rule 3. if $GIS[i] = b \wedge prev.GIS[i] \neq b$ ($b \in \{0,1\}$) then $cf[i] = 1$

A PGIS is generated when the formula F: $\sum_{i=1}^{\#FFs}(cf[i] \geq MinVal) \geq cf_limit$ becomes true. Here $MinVal$ (which replaces the constant $weight$ in T3 [4]) $= (0.75 \times Max(cf))$; $Max(cf)$ is the maximum value of all entries of the array *cf*; value 0.75 has been experimentally chosen (its goal is to provide some margin in the selection of entries that establish the common fraction); cf_limit denotes a predefined limit, which defines the minimal number of entries in *cf* with a value larger than or equal to $MinVal$. F is evaluated when STPG has been performed for the fault. If it becomes true, a PGIS is generated: The PGIS is a copy of the last stored GIS, with those entries unassigned for which the corresponding entry in *cf* has a value less than $MinVal$. For example, consider the GISes shown in Figure 2a. Before STPG starts with the next fault, F is evaluated. Assume $cf_limit = 3$, than $\sum_{i=1}^{7}(cf[i] \geq MinVal) \geq 3$ is true; i.e., the number of entries in *cf* with value larger than 3.75 is larger than 3, and a PGIS is generated: X1X01XX.

T6: PGISes to preserve existing FF-groups. This technique generates multiple PGISes from one GIS to preserve existing FF-groups. In Section 2, FF-groups have been introduced to accelerate comparison between two states and to reduce the number of to-be-compared states. However, newly added GISes can cause two or more FF-groups to be merged into one larger FF-group. Figure 3 shows an example: Addition of GIS C in the list results in the merge of FF-groups 1 and 2; GISes A, B, and C have common as-

signed entries; hence, they have to be associated to the same FF-group. Two disadvantages of merging FF-groups can be mentioned:

1. It increases the number of GISes to be compared to a circuit state, because all GISes associated to FF-groups to be merged, become associated to the merged FF-group.
2. It increases the comparison time per GIS associated to the merged FF-group, because more FFs are part of the (merged) FF-group.

Technique T6 tries to avoid group merging, by generating PGISes for each to-be-merged group. E.g., when GIS C (see Figure 3) is stored in the list, FF-groups 1 and 2 have to be merged. PGIS 1 is generated using the assigned part of C for FF-group 1 (0X0XXX) and keeping the remaining entries unassigned. PGIS 2 is generated using the assigned part for group 2 (XXX1X1). If any of the two PGISes cannot be justified then GIS C is removed, resulting in preservation of FF-groups 1 and 2.

T7: PGISes by enumeration of the assigned part of a GIS. This technique generates PGISes from existing groups of similar GISes to find new GISes for closely related FFs, such as in BCD-counters, etc. Two reasons motivate T7: First, before an ATPG stage starts, GIS-learning [4] is performed to find (sparsely assigned) GISes. For this process, depending on the number of FFs in the circuit, PGISes are generated with one, two, or more assigned entries. Many GISes can be found in this way. However, due to time limits, for circuits that contain many FFs, only PGISes are generated with one assigned entry. Second, often closely related FFs can be found in a circuit, such as the FFs of a BCD-counter or a one-hot code. Often one or more states formed by these FFs are GISes. E.g., a BCD-counter has 6 GISes (1010 up to 1111); for a one-hot code, all states with two or more 1's are illegal.

The assigned entries of identified GISes can indicate closely related FFs. Technique T7 exploits this fact. It uses the state generator (also used by GIS-learning) on the assigned entries of an identified GIS (which has no more than a predefined number of assigned entries) to generate PGISes to be justified by the STPG. For example, if state X1XX01 is detected to be a GIS, T7 generates the 7 remaining PGISes X0XX00, X0XX01, . . ., X1XX11.

5. The legal state cache

Although often PGISes can be identified as being illegal, also many PGISes are justifiable. The latter, legal states, often are justified by states which already have been proven justifiable earlier in the ATPG process. However, this information is not stored and the STPG process continues justifying these PGISes, until the all-unknown state is reached.

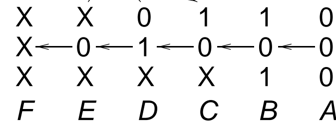

Figure 4. State A justified by state sequence $B \dots F$.

For example, Figure 4 shows a state sequence $A \dots F$. State A is being justified during GIS learning. To justify state A, a sequence of five states is generated. However, states F, E, D and C have already been proven justifiable earlier on by the GIS learning process, because they have fewer assignments than state A. The repeated justification of states C, D, E and F therefore would be useless. In order to prevent this, the *Legal State cache* (LS cache) is proposed to avoid useless repetitive justifications of known legal states. Each time a PGIS has been justified, all states of the justification sequence are legal, and are stored in the LS cache. If during the justification of a PGIS, the current state covers a state in the LS cache, justification can stop, since the current state is justifiable. We use a cache to prevent the identified legal states from requiring too much memory.

A *hit-rate* is associated to each stored legal state. The hit-rate of a legal state is increased each time that a state, being justified, covers this legal state. The states in the cache are sorted on the hit-rate (highest first) and then on the number of assignments (highest first). In order to be covered by an as large as possible *legal* state space, the stored states in the cache should be highly assigned; as opposed to the ISes, which cover a larger part of the IS space when they are more sparsely assigned.

6. Experimental results

The ATPG system DAT for combinational and sequential (3-state) circuits [4][5], implemented in C++, has been extended with the techniques proposed in this paper. The experiments have been run on a Pentium Pro 200Mhz processor with 256Mb main-memory operating under Linux (Specint95: 8.09 [12]). The time limit of the total ATPG job was 4 hours. ATPG has been performed as follows:

Pre-processes: Identification of (structurally) uninitializable and unobservable signal lines, resulting in untestable faults [4], static learning [11] (time limit 15 minutes), and GIS learning [4] (time limit 1 hour). If the GIS list is complete, then during the remaining ATPG process no further attempts are made to find or optimize GISes.

ATPG: 4 stages, each using one or more STPG methods with different search strategies The first two ATPG stages have low, and the last two have higher backtrack-limits. If

during STPG, the GIS list has been doubled in size, Check-Illegal-States (CIS) is performed to remove GISes which are covered by other GISes.

Technique T4 (direct expansion) is performed during the pre-processes when the number of assignments of a GIS is 10 or higher; and during the ATPG stages when the number of assignments of a GIS exceeds 50% of the FFs. Technique T5 (common fraction of a range of GISes) is performed with $cf_limit = 3$. Technique T7 (enumerate assigned part of a GIS) is performed when a GIS has 4 or less assignments. These parameters have been experimentally determined and avoid that the ATPG system spends too much time in those techniques.

Experiment T_all is performed as described in the previous paragraphs. Column T_all in Table 1 shows the results for the ISCAS'89 [1] circuits. These results are compared to our results published in [4]: Column $T123$ shows the results with techniques T1, T2 and T3 active, and column $T0$ shows the results without any of T1...T7. The bold values in the table indicate a significant improvement for an experiment, when comparing experiment T_all with $T123$, considering the CPU times, fault efficiencies, and the number of GISes. They show that experiment T_all, which uses the newly proposed techniques, very often is significantly better than our previous versions. E.g, for circuits s382, s526, s1494, and s35932 the CPU times are reduced and for the circuits s1423, s5378, and s38584.1 fault efficiencies are improved. Note the large improvements of the fault efficiencies for the larger circuits.

For some circuits the CPU times for exp. T_all are increased (e.g., s349, s386, and s838). This is due to the new techniques which are not (or less) required for these "STPG-easy" circuits.

We noticed that for circuits with almost 100% fault efficiency, often much CPU time is spent for a very small group of hard faults (e.g., 1 fault in s382, s400, and s444). For these circuits, 100% fault efficiency can be reached with a higher backtracklimit in the last ATPG stage; at the expense of CPU time (see the footnote with Table 1)

The sizes of the GIS lists are often reduced (see bold entries in column $\#GIS$), although they cover a larger GIS space, due to the sparser assigned GISes. The smaller GIS lists often improve the performance of the ATPG, because less GISes have to be handled (e.g., see results of s1423, s5378, and s35932). Due to the smaller GIS lists and usage of cvectors, also the memory usage often is low.

The comparisons prove the effectiveness and efficiency of our approach to constrain the STPG by (improved) determination of the illegal search space. Only for 5 of the first 25 benchmark circuits, 100% fault efficiency is not yet reached.

Table 2 shows the results of DAT in comparison with HITEC [10] (HP9000 J200; 256 MB memory; SpecInt95: 4.98), MOSAIC [2] (Sun Sparc20-70Mhz; SpecInt95: 3.11) and DUST [3] (Sun 4/60; SpecInt95: 2.5); note that the CPU times in the table are *not* normalized. The table shows that DAT often reaches better fault efficiencies, sometimes combined with lower CPU costs (e.g., s382, s526, and s35932). DAT has a very hard job on circuits s820 and s832; although the complete GIS space has been identified (the number of FFs in these circuits is very small), the fault efficiencies are low. We expect that applying IS optimization on fault-dependent ISes will improve the performance for these circuits.

	DAT		HITEC [10]		MOSAIC [2]		DUST [3]	
Circuit	CPU(s)	f.e.(%)	CPU(s)	f.e.(%)	CPU(s)	f.e.(%)	CPU(s)	f.e.(%)
s208	2	100			15	88.4	5	100
s298	5	100	972	94.5	49	99.0	60	100
s344	57	100	484	97.9	9	97.1	30	100
s349	85	100	464	98.6	186	98.0	36	100
s382	1655	99.8	5472	77.7	2340	92.4	485	96.7
s386	307	100	7	100	366	93.2	106	99.5
s400	1684	99.8	4356	84.4	900	90.3	675	96.9
s420	3	100			552	90.9	23	100
s444	2060	99.8	5364	84.0	1026	89.7	754	97.3
s526	3513	99.5	20844	61.1	606	76.5	25425	90.9
s526n	3206	99.6			684	72.0	24478	92.4
s641	36	100	5	100	49	94.2	12	100
s713	175	100	7	100	71	95.0	51	100
s820	14401	91.4	210	100	408	98.1	25282	95.4
s832	14402	92.0	345	100	1146	97.7	25681	95.6
s838	209	100			1782	82.1	71	99.7
s1196	7	100	6	100	11	100	17	100
s1238	8	100	8	100	18	100	25	100
s1423	14401	40.7	50040	48.6	1092	69.8		
s1488	4776	97.7	990	99.9	1486	98.9	25643	93.7
s1494	5310	98.2	575	99.9	2028	94.2	25252	95.3
s5378	14402	91.6	66240	74.9	4320	75.9	8881	89.6
s9234	330	100			0.1	100	3	100
s13207	14404	95.6			4680	97.6	3332	98.4
s15850	4694	100			246	99.9	5796	99.9
s35932	6149	99.9	17028	99.4	19008	99.7		

Table 2. Comparison of results of DAT, HITEC, DUST and MOSAIC.

7. Conclusions

An ATPG system (DAT) for sequential circuits has been presented, which focuses on discovering the Global Illegal State (GIS) space, to constrain the search space. The GIS space is gradually discovered implicitly during STPG for faults, and explicitly during the processes GIS learning, EIS and CIS. These processes use proposed techniques to expand the known GIS space. Given a current GIS list, they generate PGISes, which are tried to be justified using the STPG. A justification failure for a PGIS results in a new GIS, which covers GISes from the original list; they are removed. This reduces the size of the GIS list and expands the known GIS space. The four techniques proposed to expand the known GIS space are:

1. Direct expansion of new GISes (T4)
2. Generate PGISes by determining the common assigned fraction of a range of GISes (T5).

		Exp. T_all (use T1, T2, T4, T5, T6, T7 and LS cache)						Exp. $T123$ [4] (use T1, T2 and T3)			Exp. $T0$ (no techniques)	
Circuit	#nflts	#tst	#unt	seql	#GIS	CPU(s)	f.e.(%)	#GIS	CPU(s)	f.e.(%)	CPU(s)	f.e.(%)
s208	215	137	78	148	7	**1.67**	100	7	6.68	100	6.16	100
s298	308	265	43	173	**25**	**5.16**	100	29	18.30	100	18.65	100
s344	342	329	13	101	**12**	**56.82**	100	484	89.42	100	360.7	100
s349	350	335	15	139	**13**	84.95	100	134	**70.26**	100	617.2	100
s382	399	364	34	2009	161	**1655.19***	**99.75***	27	5327	99.5	5628	99.3
s386	384	314	70	294	20	307.43	100	10	**263.7**	100	433.1	100
s400	424	382	41	2008	161	**1683.80***	99.76*	27	4743	99.8	13613	97.8
s420	430	179	251	170	15	**3.26**	100	15	25.18	100	26.34	100
s444	474	424	49	1712	161	**2060.05***	99.79*	27	4372	99.8	5747	99.4
s526	555	451	101	6842	189	**3513.36**	**99.46**	27	9019	99.3	14402	97.3
s526n	553	452	99	6071	191	**3205.85**	**99.64**	27	9019	99.5	14402	97.3
s641	467	404	63	170	22	**35.68**	100	26	931.6	100	8117	99.8
s713	581	476	105	112	42	**175.11**	100	26	879.3	100	1403	100
s820	850	761	16	657	6	14401.45	**91.41**	6	14403	90.2	14402	79.9
s832	870	767	33	616	6	14402.04	**91.95**	6	14402	88.7	14401	87.8
s838	857	254	603	128	31	209.18	100	31	**101.8**	100	106.0	100
s1196	1242	1239	3	354	11	**6.59**	100	11	15.60	100	10.42	100
s1238	1355	1283	72	359	11	**7.51**	100	11	25.10	100	19.52	100
s1423	1515	597	19	103	**110**	14401.80	**40.66**	17730	14403	32.4	14403	32.4
s1488	1486	1426	26	1623	5	**4776.19**	**97.71**	5	14402	93.4	14402	93.4
s1494	1506	1441	38	1947	5	**5310.27**	**98.21**	8	14403	94.1	14402	94.0
s5378	4603	3555	660	730	**185**	14401.64	**91.57**	11912	14401	85.3	14401	80.8
s9234	6927	18	6909	3	10	329.75	100	0	**86.23**	100	65.56	100
s9234.1	6927	470	3194	40	**321**	14402.69	**52.89**	17654	14403	49.0	14403	49.2
s13207	9815	641	8738	319	3328	14404.31	**95.56**	41	14403	92.3	14401	94.2
s15850	11725	85	11640	11	**24**	4693.99	100	8753	**889**	100	801	100
s15850.1	11725	4323	1373	1270	1053	14401.38	**48.58**	1059	14402	40.9	14401	37.2
s35932	39094	35096	3984	247	**95**	**6148.74**	**99.96**	1472	14401	92.9	14403	92.9
s38417	31180	1316	900	823	**7**	14402.06	**7.11**	3039	14402	3.0	14402	3.0
s38584	36303	6614	6155	333	**654**	14401.61	**35.17**	10082	14472	28.4	14410	28.4
s38584.1	36303	16426	1666	759	**293**	14401.93	**49.84**	15200	14441	20.8	14403	20.8

#nflts: total no. of faults #tst: no. of faults detected #unt: no. of untestable faults seql: test sequence length
CPU(s): CPU usage in seconds f.e.(%): total fault-efficiency #GIS: number of GISes after ATPG has been finished
∗ For circuits s382, s400, and s444 100% fault efficiency has been reached after 7532, 7157, 9352 CPU seconds respectively

Table 1. Results of sequential circuit ATPG for ISCAS'89 circuits.

3. Generate PGISes to preserve existing FF-groups (T6).
4. Generate PGISes by enumerating the assigned part of a GIS (T7).

To accelerate access and comparison of GISes, FF-groups and cvectors have been proposed. Cvectors also reduce the storage requirements for the GISes up to a factor 10.

In order to speed-up the justification process of PGISes, the Legal State cache (LS cache) has been proposed to avoid useless repetitive justification of known legal states.

The experimental results demonstrate that: Significant improvement in computing times as well as fault efficiencies for many ISCAS'89 (and industrial) circuits has been obtained, compared to results of DAT in [4], HITEC, MOSAIC and DUST. This proves that the proposed techniques are effective and efficient.

Finally, for future extensions we intend to further improve GIS identification and expansion. Furthermore, we expect that for very hard faults, the use of the IS identification and expansion techniques in the *fault-dependent* state space, will yield significant test generation improvements.

References

[1] F. Brglez et al. Combinational profiles of sequential benchmark circuits. *Proc. of International Symposium on Circuits and Systems*, pages 1929–1934, May 1989.

[2] A. Dargelas, C. Gauthron and Y. Bertrand. Mosaic: a multiple-strategy oriented sequential atpg for integrated circuits. *Proc. of ED & TC*, pages 29–36, March 1997.

[3] N. Gouders and R. Kaibel. Advanced techniques for sequential test generation. *Proc. of ETC*, pages 293–300, 1993.

[4] M.H. Konijnenburg, J.Th. van der Linden, A.J. van de Goor. Sequential test generation with advanced illegal state search. *Proc. of ITC*, pages 733–742, November 1997.

[5] M.H. Konijnenburg, J.Th. van der Linden and A.J. van de Goor. Compact test sets for industrial circuits. *Proc. of 13th VLSI Test Symposium*, pages 358–366, 1995.

[6] H.-C. Liang, Chung Len Lee and Jwu E. Chen. Invalid state identification for sequential circuit test generation. *Proc. of Asian Test Symposium*, pages 10–15, 1996.

[7] D. E. Long, Mahesh A. Iyer and Miron Abramovici. Identifying sequentially untestable faults using illegal states. *Proc. of 13th IEEE VLSI Test Symposium*, pages 4–11, 1995.

[8] T. E. Marchok, Aiman El-Maleh, Wojciech Maly and Janusz Rajski. Complexity of sequential atpg. *Proc. of European Design and Test Conference*, pages 252–261, 1995.

[9] T. Niermann and J.H. Patel. Hitec: A test generation package for sequential circuits. *Proc. of European Design Automation Conference*, pages 214–218, February 1991.

[10] E. M. Rudnick, Janak H. Patel. Combining deterministic and genetic approaches for sequential circuit test generation. *Proc. of Design Automation Conference*, June 1995.

[11] M. Schulz, E. Trischler and T.M. Sarfert. Socrates: A highly efficient automatic test pattern generation system. *Proc. of International Test Conference*, pages 1016–1026, 1987.

[12] Specint95. The standard performance evaluation corporation. *Website: http://open.specbench.org/.*