Constraint Programming and Local Search with Multi-valued Logics for Optimisation of Test Patterns

Francisco Azevedo & Pedro Barahona {fa,pb}@di.fct.unl.pt
Departamento de Informática, Universidade Nova de Lisboa, Portugal

Abstract. This paper shows, for test pattern optimisation, that adapting constraint propagation with results obtained from local search outperforms the use of each of these techniques alone. We present a specialised constraint solver that uses a model based on multi-valued logics and incorporates such adaptation. Results are significantly better than those obtained with a highly efficient tool based on an integer linear programming (ILP) formulation on a propositional satisfiability (SAT) model.

1. Introduction

Test pattern optimisation (TPO) consists in finding, for a given fault in a circuit, a test pattern with the maximum number of unspecified inputs. Static test set compaction can later be applied to the optimised tests. After being addressed with a completely heuristic approach [3], a first formal model of TPO was proposed for minimising the number of specified PI assignments in TG for SSFs in combinational circuits, together with a tool (MTP) to solve TPO using an ILP formulation on a SAT model for TG [2].

The branch and bound procedure (B&B), available in most Constraint Programming (CP) systems, finds an optimal solution by systematically searching for better solutions. In this paper we present a formalization of the TG problem that uses a 5-valued logic [1], which explicitly considers an unspecified input as a value of the TG problem that uses a 5-valued logic [1], which outperforms MTP.

Models are not complete as shown in Fig. 2 for PI b stuck-at-1: 5-valued logic does not yield a valid output and 9-valued logic does not yield a 0/1 output.

In LS we do not construct a solution as in CP, but simply test whether an input pattern is a solution. Hence, rather than adopting a logic that encodes 2 circuits, we may simply test the 2 circuit models and check whether a PO takes value 0 in one of them and value 1 in the other.

To detect that two unspecified values come from the same source but have opposite values, unspecified bits are represented by a pair of values, where id denotes the source of the unspecified bit and denotes the parity of the signal, i.e. whether it has been subject to an even (0) or odd (1) number of inversions. Basic logic operations are described in Fig. 3 for the not- and and-gates (others are similar).

When two unspecified values coming from different sources id and id meet at an and-gate, its output Z is also unspecified and depends on both input. We thus encode it as id id (source Z and inversion parity 0).

Fig. 3. Extended logic

2. Multi-valued Logics

To implement a specialised constraint solver for the constructive phase of B&B we adopted a logic encoding the normal and faulty circuits. The combination of the possible 3 values (Boolean 0/1 plus the unspecified x) in 2 circuits defines a 9-valued logic (Fig. 1). Since values 1/0 and 0/1 guarantee different behaviour of a normal and faulty circuit, a test pattern must yield one such value at some PO. To allow a CP tool to find solutions for the TG problem with explicit x values in the input vectors, we used a simplified 5-valued logic..

### Normal

<table>
<thead>
<tr>
<th>Normal</th>
<th>0</th>
<th>1</th>
<th>x</th>
<th>0</th>
<th>1</th>
<th>x</th>
<th>0</th>
<th>1</th>
<th>x</th>
</tr>
</thead>
</table>

### Faulty

| Faulty | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |

### 9-value

| 9-value | 0/0 | 1/0 | x/0 | 0/1 | 1/1 | x/1 | 0/x | 1/x | x/x |

### 5-value

| 5-value | 0 | d-1 | x | d-0 | 1 | x | x | x | x |

### Fig. 1. 5- and 9-valued logic for normal & faulty circuits

Models are not complete as shown in Fig. 2 for PI b stuck-at-1: 5-valued logic does not yield a valid output and 9-valued logic does not yield a 0/1 output.

### Fig. 2. Circuit modelling with 5- and 9-valued logic

### 2.1 Extended Logic for Local Search

In LS we do not construct a solution as in CP, but simply test whether an input pattern is a solution. Hence, rather than adopting a logic that encodes 2 circuits, we may simply test the 2 circuit models and check whether a PO takes value 0 in one of them and value 1 in the other.

To detect that two unspecified values come from the same source but have opposite values, unspecified bits are represented by a pair of values, where id denotes the source of the unspecified bit and denotes the parity of the signal, i.e. whether it has been subject to an even (0) or odd (1) number of inversions. Basic logic operations are described in Fig. 3 for the not- and and-gates (others are similar).

### Fig. 3. Extended logic

When two unspecified values coming from different sources id and id meet at an and-gate, its output Z is also unspecified and depends on both input. We thus encode it as id id (source Z and inversion parity 0).

Fig. 4 represents the previously shown normal and faulty circuits with input vector t=x, modelled with this logic. The normal circuit outputs 0, while in the faulty
circuit (with PI b buffer stuck-at-1) the output is 1 due to the conflicting unspecified values at the or-gate. Hence, i indeed detects fault f = b stuck-at-1, since the output values for the two circuits are different and specified.

\[
\begin{array}{c|c|c|c|c|c|c}
\hline
a & 0 & 1 & 0 & 0 & 0 & 0 \\
\hline
b & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
\end{array}
\]

Fig. 4. Normal and faulty circuits with extended logic

This extended logic detects more solutions than the 5- and even the 9-valued logic. Moreover, it is very simple to use it to test alternative solutions, namely those obtained by unspecifying one bit of some already known solution.

2.2 Keeping Dependencies

To denote dependency on specified PI values, we encode circuit signals as \( \text{Set:Value} \) pairs, where Value is either a Boolean value or an unspecified value in the form \( \text{id-p} \). Set is a set of \( \text{id/V} \) pairs, denoting that turning PI id unspecified makes the signal to take a different value V (either Boolean or unspecified). Conversely, if for some PI id, there is no member \( \text{id/V} \) in Set, then making id unspecified does not affect the signal.

In Fig. 5, we show the result of keeping dependencies on the circuit previously shown. The values for the normal and faulty circuits are shown, respectively, above and under each line. Since the output of the circuit takes values 0 and 1 for the normal and faulty circuit, the fault is indeed detected with input pattern \( ab=00 \). Moreover, since a pair \( a/V \) is not present in any of the dependency sets, PI a may safely be made unspecified, and the output of the improved input pattern \( ab=x0 \) would still be 0/1.

\[
\begin{array}{c|c|c|c|c|c|c}
\hline
\text{Set:Value} & \text{Value} \\
\hline
\text{a/0:0} & 0 \\
\text{a/1:1} & 1 \\
\hline
\text{b/0:0} & 0 \\
\text{b/0:1} & 1 \\
\hline
\end{array}
\]

Fig. 5. Local search logic: PI a may be unspecified

3. Experimental Results and Conclusions

We tested our method with the ISCAS circuits [4]. Table 1 shows results obtained by using as starting point the solutions provided by Atalanta [5], a widely used TG tool. F: number of faults to test, \%X: number of PIs that are left unspecified, as a fraction of the total number PI*F. Assuming that resources spent in testing the circuit are proportional to the number of specified bits, we show the \textit{Gain} (in %) for MTP and Maxx wrt Atalanta, i.e. the saving of resources that can be achieved with the solutions provided, as well as their difference, \textit{Diff}. Average time (in seconds) spent per fault in each tool, \textit{t/f}, is also shown.

Both MTP (allowing 100 backtracks per fault) and Maxx (with a 7 seconds limit for each B&B try), increase significantly the number of unspecified bits.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>F</th>
<th>%X</th>
<th>MTP Gain t/f</th>
<th>Maxx Gain t/f</th>
<th>Diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>c432</td>
<td>36</td>
<td>524</td>
<td>56.2</td>
<td>60.8</td>
<td>33.7</td>
</tr>
<tr>
<td>c499</td>
<td>41</td>
<td>758</td>
<td>17.1</td>
<td>18.7</td>
<td>1.6</td>
</tr>
<tr>
<td>c800</td>
<td>60</td>
<td>942</td>
<td>82.2</td>
<td>83.8</td>
<td>1.6</td>
</tr>
<tr>
<td>c1355</td>
<td>41</td>
<td>1574</td>
<td>13.3</td>
<td>13.7</td>
<td>0.4</td>
</tr>
<tr>
<td>c1908</td>
<td>33</td>
<td>1879</td>
<td>44.7</td>
<td>48.4</td>
<td>3.7</td>
</tr>
<tr>
<td>c2670</td>
<td>233</td>
<td>2747</td>
<td>92.0</td>
<td>92.4</td>
<td>0.4</td>
</tr>
<tr>
<td>c3540</td>
<td>50</td>
<td>3428</td>
<td>74.6</td>
<td>77.3</td>
<td>2.7</td>
</tr>
<tr>
<td>c5315</td>
<td>178</td>
<td>3350</td>
<td>92.6</td>
<td>92.9</td>
<td>0.3</td>
</tr>
<tr>
<td>c6288</td>
<td>32</td>
<td>7744</td>
<td>22.2</td>
<td>22.2</td>
<td>0.0</td>
</tr>
<tr>
<td>c7552</td>
<td>207</td>
<td>7550</td>
<td>86.9</td>
<td>86.9</td>
<td>0.0</td>
</tr>
</tbody>
</table>

\[\text{Maxx} \rightarrow \text{MTP} \rightarrow \text{Diff}\]

The \textit{Gain} obtained with Maxx is always significantly better than with MTP, with similar time spent per fault (MTP in a SUN Sparc, 166MHz / 384 Mb, Maxx in a Pentium III, 500MHz / 256Mb).

In another experiment, taking MTP solutions as starting points for Maxx and MTP (increasing the number of backtracks per fault to 1000), Maxx always achieved better improvements than MTP, with gain difference ranging from 2 to 18 (average: 12), and significantly faster, as Maxx always took around 7 secs per fault, whereas MTP took from 20s in the smaller circuits to 72s in c6270 (for larger circuits MTP produced no results).

The model used in the CP step, although less complete than that used in the LS step, was often able to provide a different starting point that enabled LS to escape from local optima. Moreover, the extended logic used for LS proved quite useful to improve a solution found by constraint propagation.

This paper has shown that using both CP and LS to solve constraint optimisation problems may outperform using each of these techniques alone. In TPO, the 2 interacting approaches used different multi-valued logics that we adopted and developed. Constraint propagation uses a simpler logic that misses possible solutions, but for which an efficient solver was developed. A more expressive logic could nevertheless complement the previous one and be used efficiently in LS.

References