Deductive Diagnosis of Digital Circuits*

J. J. Alferes[1], F. Azevedo[1], P. Barahona[1], C. V. Damásio[1], and T. Swift[2]

[1] Centro de Inteligência Artificial (CENTRIA), FCT/UNL
2829-516 Caparica, Portugal. {jja|fa|pb|cd}@di.fct.unl.pt
[2] Dept. of Computer Science, State University of New York
at Stony Brook, NY 11794-4400, USA, tswift@cs.sunysb.edu

Abstract. In this paper we present an efficient deductive method for addressing combinational circuit diagnosis problems. The method resorts to bottom-up dependencies propagation, where truth-values are annotated with sets of faults. We compare it with several other logic programming techniques, starting with a naïve generate-and-test algorithm, and proceeding with a simple Prolog backtracking search. Two approaches using tabling are also studied, based on an abductive approach. For the sake of completeness, we also address the same problem with Answer Set Programming. Our tests recur to the ISCAS85 circuit benchmarks suite, although the technique is generalized to systems modelled by a set of propositional rules. The deductive method outperforms others by orders of magnitude.
Keywords. Fault Diagnosis, Logic Programming, Tabling

Introduction

Because of its simplicity and applicability, model-based diagnosis has proven an important problem in artificial intelligence. Simply put, model-based diagnosis can be seen as taking as input a partially parameterized structural description of a system and a set of observations about that system. Its output is a set of assumptions which, together with the structural description, logically imply the observations, or that are consistent with the observations. This corresponds to the *Matching-Abnormal-Behaviour* (MAB) diagnosis conceptual model of [10].

A problem-solving system for model-based diagnosis can be used to diagnose faulty behaviour of systems from their specifications, and may be valuable in the manufacture of electrical circuits, engine components, copier machines, etc. However, such a system could also be used to allow deliberative agents revise their plans, to parse natural language in the presence of errors and other tasks.

As stated, model-based diagnosis bears a strong resemblance to satisfiability in first-order logic. Accordingly, the implementation of model-based diagnosis has most often been based on general problem-solving systems, such as truth maintenance systems (TMS) or belief revision systems. However, given its logical flavour, model-based diagnosis should be amenable to logic programming (LP) techniques, such as abduction or default logic. For diagnosis, the power of an LP approach, if it can be

* This work was partially supported by Praxis XXI Project TARDE.

made successful, is that the search of problem space can be made by an optimized general purpose engine rather than using a specially designed diagnoser or TMS.

In this paper we explore various LP approaches to model-based diagnosis, and apply them to the *c6288* digital circuit from the ISCAS'85 set of benchmarks [9]. *C6288* (**Fig. 1.**) is a 16-bit multiplier, which can be seen as a grid-like pattern of 240 half and full adders, consisting of 2406 logical gates in all.

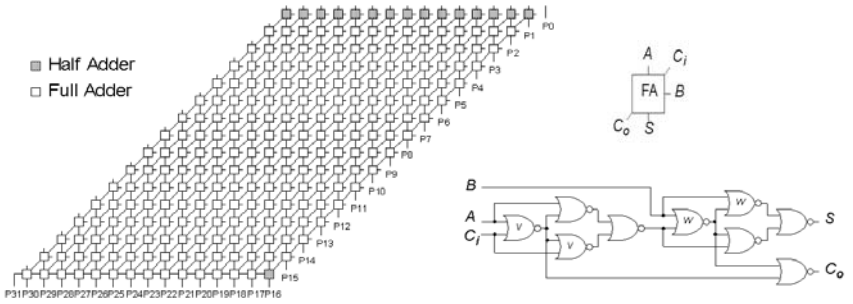

Fig. 1. High-level Model of *c6288* Multiplier Circuit and full adder module (images obtained from http://www.eecs.umich.edu/~jhayes/iscas/).

C6288 is of special interest in that it has traditionally proven difficult to diagnosis system. In [6], it is reported that several special-purpose diagnosers could not reliably detect faults in this circuit. However, the best LP approaches reliably detect all faults, and appear superior in the execution times. Moreover, LP approaches, when compared to the special-purpose ones, also have the advantage of requiring a very small amount of code – often less than a hundred lines.

In our experiments, we adopt the usual *stuck-at* fault model, where faulty circuit gates can be either *stuck-at-0* or *stuck-at-1*, respectively outputting value 0 or 1 independently of the input. This paper first presents two naïve approaches that use a generation mechanism to identify possible sets of faults and then a test mechanism to test whether the faults are consistent with a given set of observations. We then present approaches that use tabling to abduce faults consistent with observations. As an alternative to abduction, we present an approach based on generating diagnoses as a stable model (SM) of a program, an approach that uses a novel mechanism of grounding the input to the SM generator via tabling. We next show how a deductive technique can be adapted to efficiently derive diagnoses and be advantageously implemented in Prolog. Finally, we compare the performance of all techniques and analyse their strengths and weaknesses. We note that programs discussed in this paper can be obtained via http://www.cs.sunysb.edu/~tswift/interpreters.html.

Generate and Test

Perhaps the simplest, even naïve, method to test a circuit for a diagnosis is a simple generate-and-test method. Essentially, for each faulty gate that we intend to test, we simply replace its model by fixing its output to the appropriate faulty value which, in our running example is the negation of the correct value – generate phase. We then compare the output produced by the faulty model with the observed output. If they are the same, the (possible) fault is accepted, otherwise rejected – test phase.

To implement this approach we use a *circuit(+InputVector, +OutputVector, +Diagnosis)* predicate, which is a huge clause corresponding to a bottom-up evaluation of the circuit gates. The circuit topology is implicitly represented by the position of variables in gate goal calls in the body of *circuit/3*. The values of nodes in the circuit are propagated through usual logical Prolog variables (each variable models one node). As an example, the rule for a NOR gate is:

```
nor(I1, I2, Out, Name, Faults):-  member(Name,Faults) -> or(I1,I2,Out) ; nor(I1,I2,Out).
```

Faults is the current diagnosis (a list of faults) to test. The predicates *or/3* and *nor/3* are the truth tables for the appropriate logical connectives. The normal output of a gate is negated in order to simulate its incorrect behaviour. The fault mode of the gate can be determined immediately from its output value.

This code can be run in two ways. In a classical *generate-and-test*, the input is fixed and the set of faults is iterated over all the possible diagnoses. The output for each set of faults (a diagnosis) is then compared with the observed output vector. If identical, then a diagnosis has been found. Another method consists in forcing the output vector in the call of *circuit/3*. This is more efficient, since the predicate can fail immediately on an output bit generation. We call this approach *generate-and-check*.

In the worst case, the complexity of the methods discussed above is $O(G*F)$ where G is the number of gates of the circuit and F the number of faults, since the behaviour of all the circuit gates is checked for each fault. For single faults, the worst case complexity is thus $O(G^2)$. Of course, worst case complexity for n faults is $O(G^{n+1})$.

Backtracking

An alternative to the previous generate-and-test is a backtracking approach, making use of the in-built depth-first search strategy of Prolog engines. Instead of taking as input the faulty state of the gate, this approach simply models all the possible states of the gate, and returns either a list with the faulty state or an empty list. The last argument of *circuit/3* returns the list of faulty gates. The use of free variables allows us to fix the maximum number of faults in a diagnosis at call time, and propagate this information throughout the circuit. The NOR-gates are now implemented as follows:

```
nor(I1, I2, Out, _, Faults, Faults):- nor(I1,I2,Out).
nor(I1,I2, Out, Name, [Name|Faults], Faults):- or(I1,I2,Out).
```

The first rule captures a correct NOR-gate, and so does not change the current list of faults. The second rule implements a faulty NOR. It simply returns the output of an OR gate, unifying and consuming the name of the gate in the list of faults. Each gate is thus a choice point, with alternative normal and abnormal behaviours.

The approach is general, in the sense that the number of faults intended may be imposed in the call. For example, a call to goal *circuit(Input, Output, [F1,F2])* only succeeds in circuits with a double fault. We obtain the collection of all single faults *F* by backtracking goal *circuit(Input, Output, [F])*, e.g. with a *findall/3* meta-predicate.

The asymptotic complexity of this approach is similar to generate-and-check, i.e. *$O(G^2)$* for single faults. However, given its better use of logical variables in Prolog, tends to detect inconsistencies earlier and execution is significantly faster in practice.

Tabling Approach

In this section we present a solution to the problem of circuit fault diagnosis based on tabled abduction implemented in XSB [14]. The use of tabling is natural for handling the grid-like structure of *c6288*, which can require a huge amount of recomputation of circuit values, if a top-down approach is used. For tabling, the circuit topology is represented by sets of facts stating the gate types and connections.

In the tabling approach we represent the faulty behaviour of a gate as an abducible hypothesis, and a diagnosis as an abductive context, i.e. as a set of abducibles. The rules for the normal behaviour of a gate know the maximum number of faults that may be abduced in the calling context, given as an ordered list of gates already assumed faulty, thus ensuring tractability. Upon success, we get the abductive context used to prove the output value.

If, e.g., an AND-gate is not faulty and one of its inputs is zero, then the output is 0, and abductive contexts do not need to be manipulated within the rule. However, in order to obtain a 1 in the output of an AND-gate, both inputs must be 1. But in order to prove that the first input is 1, abductions may need to be performed *on both inputs*. Accordingly, the rule maintains not only the full abductive context at each point in its evaluation but also the set of abducibles derived in the rule itself. At each stage, if faults are abduced, the maximal number of allowed abducibles is reduced. In addition, it must be ensured that the abductive context obtained after determining the second input value is consistent with what is required to obtain the first input value. This is done with a re-evaluation of this first input value, setting the maximal number of abducibles to 0, meaning that no abductions can be performed on this testing phase.

The rules for the abnormal behaviour follow the same structure. The major difference is the use of an *abduce* predicate that performs the abduction, if possible. If abduction is performed, the abductive context is updated with the corresponding gate. Gate predicates are tabled, avoiding redundant computation of values and abductions.

This implementation is substantially optimised when determining single faults only. An important improvement is the use of the list of inputs sorted according to the heuristic principle that values that disagree with the ones of the normal behaviour should be evaluated (or corrected) first. The same idea is applied to the output vector.

Since we are only interested in single faults, the rest of the circuit should be working correctly. Therefore, the values of nodes in the model of the circuit's correct behaviour can be reused to determine the input values of the abnormal gate. This can be further generalized. In fact, we check first whether a gate has been abduced faulty. Then, we only have to recalculate the values of the input gates whenever these depend on the faulty gate. This technique requires a dependency checking. Again, tabling is a suitable programming technique for finding out dependencies, since it reduces to computing the transitive closure of the connection graph. The dependency checking combined with the ordering of inputs seems to payoff in most of the cases.

Stable model programming

A new, and growing in importance, LP paradigm is that of Stable Models (or Answer-set) Programming [11,12], for which several implementations already exist [4,13]. In it, solutions to a problem are represented by the stable models [7] of the corresponding program, rather than by answer substitutions of a single model of the program, as in traditional LP. In traditional LP (as in the other approaches we present) the diagnoses of a circuit are represented by terms, the clauses of the program being viewed as their recursive definition. In stable model (SM) programming, clauses are viewed as constraints on the diagnoses, each diagnosis being represented by a model of the program defined in terms of those constraints. As claimed in [11], *"stable model programming is especially well suited for all problems where solutions are subsets of some universe, as each solution can be modelled by a different stable model"*. This is definitely the case of circuit diagnosis, where solutions are subsets of abnormal gates, and so we have also used this new approach to solve the circuit problem.

To represent our circuit diagnosis problem in SM programming, all we have to define is a suitable set of constraints over the predicates that define the circuit and the diagnoses. An important constraint in this domain is that each gate is either normal or abnormal, and cannot be both. This is easily coded by the following two rules:

```
abnormal(X) :- gate(X), not normal(X).
normal(X) :- gate(X), not abnormal(X).
```

i.e. if *X* is a gate that is not normal then it must be abnormal, and vice-versa.

Moreover, if some, e.g., AND-gate is normal, then its output value must be the conjunction of its inputs. If it is abnormal, its output is the negation of the conjunction. Constraints imposing exactly this are:

```
val(out(and2,G), V):- normal(G), val(in(and2,G,1), I1), val(in(and2,G,2), I2), and(I1,I2,V).
val(out(and2,G), V):- abnormal(G), val(in(and2,G,1), I1), val(in(and2,G,2), I2), nand(I1,I2,V).
```

It remains to be imposed that: *a)* no point can simultaneously have 2 different values; *b)* all values of a given output vector are observed; and *c)* only single-faults occur:

```
 inconsistent :- val(P,V1), val(P,V2), V1 \= V2.
explains :- val(out(and2,c545gat),V1), ..., val(out(nor2,c6288gat),V32).
nonsingle :- abnormal(G1), abnormal(G2), G1 \= G2.
goal :- explains, not inconsistent, not nonsingle.
:- not goal.
```

where each *Vi* in the *explains/0* clause is replaced by the output value of the corresponding gate in a given output vector. This clearly resembles the formulation of an abduction problem: our goal is that all observed output values are explained, there are no inconsistent and no diagnoses with more than one abnormal gate. And we are only interested in SM's in which our goal is satisfied (i.e. it is not false). The SM's of this program, restricted to predicate *abnormal/1*, exactly correspond to the diagnosis of the circuit.

For computing the SM's of the described program (i.e. the single-fault diagnoses of the circuit) we have used the *smodels* system [13] version 2.26 for Windows[2]. For

2 An alternative is to use the DLV system with a diagnosis front-end [3].

dealing with the grounding of the program, required by *smodels*, and for pre-processing away the function symbols *out/2* and *in/3* used in the representation of the circuit, we have developed an XSB-Prolog program. Although this is all that is required of the XSB-Prolog for this circuit diagnosis problem, the program does more. The additional functionality of the mentioned XSB-Prolog program might be crucial for other diagnosis problems and, in general, for other problems that can be coded as abduction.

Our XSB-Prolog program starts by running the query *goal* in a program with the representation of the problem having the above clause for *goal/0*, without the last clause (*:- not goal*), and where all predicates are tabled. In this execution, all calls that depend on loops over negation are suspended. Note that, in the above representation, the only loop over negation is the one between predicates *abnormal/1* and *normal/1*. After the execution, the tables of the various predicates contain the so-called *residual program* [2], which has, for each predicate, the (non-failing) rules used during execution, simplified by removing all body literals proven true (and not suspended). It is well known [2] that the SM's of the residual program are the same as those of the part of the original program relevant to the query. Moreover, if all the calls during the execution are ground (which is the case for our representation) the residual program is also ground. Thus, this residual program, after some simple pre-processing that eliminates the function symbols *out/2* and *in/3*, is what is passed to *smodels* for computing the diagnoses. This method is now easy to implement, due to the recent XAsp package of XSB 2.6, which already provides special predicates for this purpose and linking to *smodels.* Its main advantage over a direct usage of *smodels* is that only the part of the program relevant to the query has to be considered when computing the SM's. Besides gains in efficiency, this is also important for general abduction problems: in this way, the obtained abductive solutions come only in terms of abducibles relevant to the query (i.e. only relevant abductive solutions are computed).

Deductive Diagnosis

As an alternative to model the circuit diagnosis problem, we represent digital signals with sets and Booleans. A deductive technique [1,8] is used to simulate the circuit behaving normally, as well as to deduce the behaviour of all faulty circuits. This technique has been used by the Electronic CAD community for fault simulation, but in this section we show how to apply it to our diagnosis problem.

Since the faulty behaviour of a circuit can be explained by several sets of faults, we represent a signal not only by its normal value but also by the set of diagnoses it depends on. More specifically, a signal is denoted by a pair *L-N*, where *N* is a Boolean value (representing the Boolean value of the circuit when there are no faults) and *L* is a set of diagnoses, that might change the signal into the opposite value. For instance, for single faults, *X={g/0,i/1}-0* means that signal *X* normally is *0* but if gate *g* is stuck-at-0, or gate *i* is stuck-at-1, then its actual value is 1. *∅-N* represents a signal with constant value *N*, independently of any fault. For generality, we need to explicitly represent the fault modes in the set of faults. In the following we assume that any gate in a circuit may be faulty.

A gate *g*, that can either be normal, stuck-at-0 or stuck-at-1, may be modelled by means of a normal gate to which a special buffer, an **S-buffer**, is attached to the output. As such, all gates are considered normal, and only S-buffers can be faulty. The modelling of S-buffers is:

In	$\varnothing$-0	$\varnothing$-1	L_i-0	L_i-1
Out	{g/1}-0	{g/0}-1	$\{g/1\} \cup L_i$ - 0	$\{g/0\} \cup L_i$ - 1

When the input is 0 and independent of any fault, the S-buffer output would normally be also 0, but if it is stuck-at-1 then it becomes 1. More generally, if the normal input is 0 but dependent on L_i, the output depends not only on *g/1* but also on input dependencies L_i. The same reasoning can be applied to the case where the normal input signal is 1, and the output of an S-buffer *g* with input L_i-N can be generalised to $\{g/\overline{N}\}\cup L_i$-$N$, where $\overline{N}$ stands for the complement of Boolean value *N*.

Normal gates fully respect the Boolean operation they represent. We discuss the behaviour of NOT and AND-gates as illustrative of these gates. All other gates can be modelled as combinations of these. Given the above explanation of the encoding of digital signals, for a normal NOT-gate whose input is signal *L-N*, the output is simply L-$\overline{N}$, since the set of faults on which it depends is the same as the input signal.

For an AND-gate, in the absence of faults, the output is the conjunction of the normal inputs. The set of faults that may change the output signal into the opposite of the normal value is less straightforward to determine. When both normal inputs are 1 (1st case of the table below), a fault in set L_1 or set L_2 justifies a change in the output. In the second case (two 0s), to invert the output signal, a fault in both L_1 and L_2 must exist. So, the set of faults that justify a change in the gate's output is the intersection of the input sets. In the last two cases, to obtain an output different from the normal 0 value, it is necessary to invert the normal 0 input, and not to invert the normal 1 input (justifying the set difference in the output):

In1	L_1-1	L_1-0	L_1-0	L_1-1
In2	L_2-1	L_2-0	L_2-1	L_2-0
Out	$L_1 \cup L_2$-1	$L_1 \cap L_2$-0	$L_1 \setminus L_2$-0	$L_2 \setminus L_1$-0

To model the diagnosis problem and find the possible single faults that explain the faulty output vector *F*, a bit-wise comparison between *F* and the deduced simulated logic output must be performed. Let r_o and s_o denote, respectively, the real and simulated value of output bit *o*, where r_o is a Boolean value B_o and s_o is a set-Boolean pair L_o-N_o. When $B_o \neq N_o$, the only possible single faults are in L_o and any such fault explains B_o. When $B_o = N_o$, none of the faults in L_o occur. Hence, the set of faults that justify the full incorrect output vector *F* is given by $\{f\text{: } \forall_o ((B_o \neq N_o \Rightarrow f \in L_o) \wedge (B_o = N_o \Rightarrow f \notin L_o))\}$ where *o* ranges over all the output bits. The diagnostic solution is then given by intersecting all the dependency sets L_o where r_o is incorrect and removing the union of L_o where r_o is as expected.

The LP implementation of the deductive fault diagnosis is immediate: we simply propagate bottom-up the signals over the circuit (as in the generate-and-test, and backtracking approaches), making use of logical variables and unification. In contrast with the generate-and-test implementation, Boolean values are now substituted by a term *Value-ListOfGates*, and the gates' behaviours are as described above. To implement the set operations, we resorted to the *ordsets* library for operations over sorted lists of SICStus Prolog [14]. Of all the approaches in this paper, it turns out that this is the most efficient one. This is as expected, since with this approach only one pass in the circuit is needed to extract the information needed to compute all the faults for all the output vectors.

This method can be extended to handle multiple fault diagnoses. The major problem is the representation of all the possible diagnoses. In the single fault case, our sets may have at most *2*G* gates, where *G* is the number of gates in the circuit (2406 for *c6288*). However, for double faults the lists may expand to $4*G^2$ (around 23 million elements for *c6288*!). A better representation is needed in order to avoid this explosion. We tried to encode sets of double faults by sets of pairs of the form *(f,ListOfFaults)* or *(f,-ListOfFaults)*. For instance, the pair (10/1,[20/0,40/1,50/1]) represents the set of faults {(10/1,20/0), (10/1,40/1), (10/1,50/1)}, while (10/1,-[20/0,40/1,50/1]) stands for the set of all double faults, containing 10/1, minus the above ones. We have extended the ordinary set operations to pairs of this form and tested it with *c6288*. All the double faults for *c6288* could be determined in a reasonable amount of time (see the conclusions section).

Results and Conclusions

In this paper we presented several approaches to the circuit diagnosis problem, and corresponding LP implementations. The implementations were tested using the same input vector, 01001000000100010001000110100000, corresponding to the multiplication of 34834 by 1416, returning 49324944, represented in binary by (least significant bit first) 00001001110001010000111101000000.

From this correct output we flipped a bit at a time, obtaining 32 incorrect output vectors. The results for the various approaches were as shown in Table 1.

Timings should be looked with some care. Note that we are using different Prolog systems, with possible impact on the performance. In the last column of the table, only the total time appears since, for the deductive approach, the information needed to obtain the diagnoses is computed in a single propagation over the circuit (this phase takes 220ms). The diagnoses for each test are then obtained by set operations on the results, this phase taking a total of 610ms. On average, for a single test vector, diagnoses can be found in around 20ms, after propagation. Thus, total execution time reduces to 240ms. This is by far the best method presented here. The method can also be generalized to multiple faults. When computing all double faults, the propagation phase took approximately 780 seconds. Note that the operations on sets of faults are now more complex and therefore we have a 3500-fold slowdown. An implementation for larger cardinality of faults is an open research problem. Notice that a similar technique can be used in Abductive LP, widening the applications of the method.

As expected, generate-and-test takes constant time. Generate-and-check is a little better, but its performance degrades as the wrong bit becomes more and more significant, since incorrect assumptions fail later. The backtracking version performs quite well, but shows the same problem of generate-and-check, for the same reasons.

Table 1. Diagnosis results (in seconds) for 32 incorrect ouput vectors of *c6288*.

Vector	#Sols	Test[3]	Check[3]	BT[3]	Tabling[4]	Smodels[5]	Deductive[3]
1	1	19.05	0.92	0.00	0.06	5.23	-
2	9	19.67	3.41	0.33	0.22	42.13	-
3	18	19.94	4.39	0.44	0.45	47.00	-
4	27	19.11	5.44	0.66	0.73	50.61	-
5	11	19.11	6.37	0.83	0.36	53.30	-
6	45	19.50	8.08	1.20	1.44	65.00	-
7	54	20.11	9.77	1.60	1.85	72.18	-
8	23	19.05	9.94	1.81	1.08	65.86	-
9	11	19.06	10.77	2.20	1.14	67.50	-
10	11	19.28	11.81	2.63	3.40	70.67	-
11	90	20.32	12.80	3.30	7.35	126.31	-
12	80	19.06	13.73	3.79	5.46	125.97	-
13	87	19.06	15.65	4.34	7.01	140.56	-
14	10	19.23	16.04	4.89	6.91	76.44	-
15	91	20.37	16.26	5.38	10.68	157.05	-
16	21	19.06	16.42	5.60	8.92	77.63	-
17	135	19.06	16.59	6.26	14.13	225.12	-
18	127	19.23	17.90	6.54	13.95	103.67	-
19	101	20.26	16.48	5.71	11.48	136.01	-
20	104	18.95	16.59	5.71	10.81	132.96	-
21	33	19.01	16.59	5.66	8.94	81.26	-
22	31	20.43	17.96	5.71	10.22	80.13	-
23	37	19.00	16.53	5.72	11.56	86.38	-
24	33	19.01	16.59	5.71	12.73	84.83	-
25	64	19.44	16.64	5.71	15.54	87.10	-
26	25	19.99	17.96	5.77	6.92	75.09	-
27	46	19.01	16.59	5.71	7.45	73.20	-
28	37	18.95	16.59	5.77	3.42	64.35	-
29	28	20.43	16.80	6.59	2.05	57.79	-
30	19	19.00	17.91	6.31	1.07	52.32	-
31	1	19.01	16.64	5.77	0.11	9.98	-
32	10	19.33	16.59	5.77	0.51	46.61	-
Total Time		621.09	432.75	133.42	187.95	2640.24	**0.83**

The tabling approach is very good at solving problems with a small number of faults, the running times being almost independent of the wrong bit. The justification is the dynamic ordering of inputs and dependencies checking used to direct the search. It gets worse for greater number of faults since more memory is required to store the tabled predicates. Also notice that XSB Prolog is much slower than SICStus.

The SM programming approach is, among those presented, the least efficient. However, this approach has been specially tailored for solving NP-complete problems (which is not the case for our single-fault circuit diagnosis). Nevertheless, we were impressed with the robustness of the *smodels* system, which was capable of handling the very large files resulting from the residual program for each test in this circuit. In fact, for each output vector, the (ground) logic program, generated by the XSB-Prolog program, that served as input to *smodels* has, on average, 31036 clauses and around 2.4 MB of memory. On the other hand, the representation of the circuit and of the problem is, in this approach, (arguably) the most declarative and easier one.

3 SICStus 3.8.5 on a Pentium III, 750 MHz with 128 Mb of RAM, Windows 98
4 XSB-Prolog 2.2 on a Pentium III, 733 MHz with 256 Mb of RAM, Linux
5 SModels 2.26 on a Pentium III, 733 MHz with 256 Mb of RAM, Windows 98

We have also tried to implement a solution resorting to SICStus library of constraints over Booleans (clpb). Our efforts proven useless, since SICStus was unable to handle the constraints we generated (usually, ran out of memory).

Although we did not run specialized diagnosis systems in the same platform, we can compare our times with the ones presented by those systems some years ago, and take into account the hardware evolution. The results of the shown LP approaches are then quite encouraging. For example, the results of the DRUM-II specialized system presented in [5] seem worse than ours: it can take 160 seconds to diagnose all single faults in *c6288* for a specific output vector. We dare to say that this system, even if ported to an up-to-date platform, would still be less efficient than our deductive approach (which takes 0.83 seconds to produce the diagnoses for the 32 output vectors), and would possibly be comparable to our general approaches of backtracking and tabulation (which, for the worst case take, respectively, 6.59 and 15.54 seconds).

References

1. D. B. Armstrong, A Deductive Method of Simulating Faults in Logic Circuits, *IEEE Trans. on Computers*, Vol. C-21, No. 5, pp. 464-471, May, 1972.
2. W. Chen and D. S. Warren, Computation of stable models and its integration with logical query evaluation, *IEEE Trans. on Knowledge and Data Engineering*, 1995.
3. T. Eiter, W. Faber, N. Leone, and G. Pfeifer, The diagnosis frontend of the dlv system, *AI Communications*, 12(1-2):99-111, 1999.
4. W. Faber and G. Pfeifer, DLV homepage, 1996. At http://www.dbai.tuwien.ac.at/proj/dlv/
5. Peter Fröhlich and Wolfgang Nejdl, A static model-based engine for model-based reasoning, In *Proceedings of IJCAI97*, pages 466--473, 1997.
6. P. Fröhlich, *DRUM-II Efficient Model-based Diagnosis of Technical Systems*, PhD thesis, University of Hannover, 1998.
7. M. Gelfond and V. Lifshitz, The stable model semantics for logic programming, In R. Kowalski and K. Bowen, editors, *ICLP'88*, pages 1070-1080. MIT Press, 1988.
8. H. C. Godoy and R. E. Vogelsberg, Single Pass Error Effect Determination (SPEED), *IBM Technical Disclosure Bulletin*, Vol. 13, pp. 3443-3344, April, 1971.
9. ISCAS. Special Session on ATPG, *Proceedings of the IEEE Symposium on Circuits and Systems*, pages 663-698, Kyoto, Japan, July 1985.
10. P. Lucas, Symbolic Diagnosis and its Formalisation, *The Knowledge Engineering Review*, 12(2), 109-146, 1997.
11. M. Marek, M. Truszczyński, Stable models and an alternative logic programming paradigm, *The Logic Programming Paradigm: a 25-Year Perspective*, pp. 375-398, Springer, 1999.
12. I. Niemelä, Logic programs with stable model semantics as a constraint programming paradigm, In I. Niemelä and T. Schaub, editors, *Computational Aspects of Nonmonotonic Reasoning*, pages 72--79, 1998.
13. I. Niemelä and P. Simons, Smodels - an implementation of the stable model and well-founded semantics for normal logic programs, In *4th LPNMR*, Springer, 1997.
14. Programming Systems Group of the Swedish Institute of Computer Science. *SICStus Prolog User's Manual*, 1995.
15. *The XSB Programmer's Manual: version 2.1, vols. 1 and 2*, 2000. http://xsb.sourceforge.net