A Solver for Non-linear Boolean Functions

W. Mark Vanfleet, Michael Dransfield
National Security Agency
9800 Savage Road, Suite 6709
Fort George G. Meade, MD 20755-6709
E-mail: `wvanflee@radium.ncsc.mil` vspace*6mm

John Franco, Robert Price, John Schlipf, Jeff Ward, Sean Weaver
Computer Science, ECECS
University of Cincinnati
Cincinnati, OH 45221-0030
E-mail: `franco@ececs.uc.edu`

March 13, 2001

1 Executive Summary

This paper describes a tool we call State Based Satisfiability (SAT) Solver (SBSAT). SBSAT is being developed jointly by NSA, Cryptographic Evaluation Group (CEG), and the University of Cincinnati, as funded by the NSA, Information Assurance Research Organization (IARO). The primary goal of the SBSAT Solver is to combine the strengths of existing Satisfiability tools and Binary Decisions Diagram (BDD) based tools into a single unified tool which allows a user to control the solver directly from his or her native domain space. SBSAT has an input interface which pre-processes the user's problem, potentially modeled in various forms (e.g. BDDs, CNF, etc.), possibly to solution, but more often to a new representation which is amenable to selected search algorithms. Rather than translate input expressions to CNF, as is done with traditional SAT solvers, SBSAT attempts to search in a domain space as close to the original function set as possible. Doing so avoids loss or mutilation of domain-specific knowledge which is often very helpful during search, but requires a new (non-CNF) form of SAT search.

The new search paradigm is based on State Machine Branching (SMB). In order to solve problems in a user's domain space, we created what we call State Machines Used to Represent Functions (SMURFs). SMURFs contain pre-computed information about the possible future outcomes of variable assignments given any reasonable partial assignment. A single SMURF maintains such information for an entire Boolean function, not a single clause, and typically requires a relatively large amount of memory. The ability to maintain a useful collection of SMURFs has only recently become practical as even laptop computers now carry 1/2 GB of RAM routinely. Thus, we present SMURFs as an effective means to trade space for time.

Using a SBSAT prototype, problems too difficult for BDD-based solvers or CNF-based solvers have been solved reasonably well. This initial success is due, in part, to providing the user with more effective control over search in his original familure domain space. In particular, the development of heuristics which make sense for various classes of functions.

Discovering effective heuristics for SMURFs and developing effective SMB search procedures have become an important secondary goal of the research contract. We have undertaken the task of lifting all of the concepts from traditional, even state of the art, SAT Solvers to SMURFs and SMB. We have been particularly successful in doing this with heuristics (procedures for deciding which variable and value to branch on) where we are able to take advantage of the SMURF data structures and develop heuristics that are more powerful than any existing heuristics in use in other tools. We have also developed effective pruning mechanisms in SMB. Because CNF data structures can be embedded into the SMURF data structure, we must do at least as well in search-space economy as other CNF tools. But, we can often do much better because we can combine many CNF's into a single SMURF to obtain inferences that the traditional CNF SAT Solver would have taken much longer to find. We have also generalized the concept of autarkies[1] for pruning even further. Handling of autark assignments can be added relatively painlessly because information to support their discovery and use can be pre-computed prior to branching.

This brings us to another goal of the contract which is to develop data structures that allow pre-computing or memoizing anything that can be effectively pre-computed or memoized, and that can be of use in the SMB brancher. Any information that the SMB brancher may derive more than once is a candidate to be pre-computed. Because of pre-computation, we are able to efficiently bring generalized autarkies into the brancher. The literature in the CNF world is currently limited to what is known as *linear autarkies*. We are able to detect and use *generalized non-linear autarkies* in our SMB brancher. Due to pre-computation, we are able to incorporate a technique that belonged heretofore exclusively to the BDD world, namely simple existential quantification: during search, whenever a variable only occurs in one SMURF we are able to existentially quantify that variable away. In addition, we memoize *lemmas*[2] (a very promising area of research in the CNF SAT world). We are again able to generalize lemmas based on SMURFs rather than CNF's. Putting all of these techniques together we believe that we have developed a remarkably effective hybrid BDD and non-linear SAT solver. We have captured the best of both worlds. We have exceeded the capabilities that either world could have achieved on its own.

KEY WORDS: Binary Decision Diagrams, BDD, Satisfiability, SAT, State Machines, State Machines Used to Represent Functions, Smurf, Branching, State Machine Branching, SMB, State Based SAT Solver, SBSAT, Conjunctive Normal Form, CNF SAT, Non-CNF SAT, Non-Linear SAT, Davis-Putnam-Loveland Procedure, Lemmas, Heuristics, Linear Autarkies, Non-Linear Autarkies, Existential Quantification, Boolean, Logic

[1]Autarkies are defined later but to get a sense of these objects now we mention that pure literals in CNF formulas are examples of very simple autarkies.

[2]Minimal partial assignments which support no solutions.

2 Results

The particular form of formula ϕ which mainly interests us is the class of *Layered Boolean Functions* (LBF) which is described in Appendix A. An LBF imposes the following restrictions:

1. $n = m$.
2. The domain space is partitioned into segments, or layers.
3. Functions are similarly partitioned.
4. If function and input segment sizes are equal, the order of functions in a function segment uniquely corresponds to the order of variables in the corresponding input segment.
5. Both input and function segments are themselves ordered for easier bookkeeping.
6. Inputs to a particular function may come from an input segment of order lower or equal to that of the segment corresponding to the function.

A novelty of the solver is that it consists of a collection of *n state machines*, each acting on behalf of one Boolean function, and a *brancher* whose *heuristics* are aware of and exploit this state-based architecture. Since much information is precomputed in the state machines, inferences can be made more efficiently and sooner resulting in fewer backtracks and higher backtracks per second. In addition, to help reduce the number of backtracks further, the solver has the ability to save inferences for future use when they are discovered during branching. Each such inference is called a *lemma*. The primary components, namely state machines, brancher, heuristics, and lemmas, are described elsewhere.

Preliminary results indicate that SBSAT has great potential and that considerable additional research is needed to fully exploit this potential. Currently implemented features have been studied in isolation, and in cooperation with each other, and solver performance has been compared against existing state-of-the-art SAT solvers such as GRASP and SATO3.

Tests have been conducted on semi-random LBFs of equal segment sizes as well as on one actual input, called EXAMPLE (from a class of problems described in Section B). In the case of the semi-random LBFs, there are at most $k+1$ inputs for each Boolean function f, k of which (the *width*) either are random in f's input segment or consecutive (wrapping around the segment when necessary), and the remaining one, if admissible, takes f's order in the previous input segment. LBFs of the latter type are called *Sliding* and those of the former type are called *Random*. The functions for all semi-random LBFs are constructed as follows. First, randomly choose an input vector of n Boolean values and an output vector of n Boolean values. Let R be the total number of rows in all truth tables of all functions of ϕ. For $R/2$ iterations, repeatedly choose randomly one of the *unmapped* rows of ϕ until finding one that may be mapped to *true* without violating input-output consistency, and then set that row to map to *true*. When finished, set all unmapped rows to *false*. Finally, reduce the number of input mappings consistent with the chosen output vector by applying a procedure which leaves ϕ with a minimal number of consistent mappings to the given output vector.

Tests use the current SBSAT prototype software. All times are on a 450 MHz single processor Pentium III PC with 640MB of RAM. Times for SBSAT do not include preprocessing time (preprocessing times for the inputs given in the tables below are between 15 and 60 seconds).

LBF instance					**Performance**	
Type	In Vars	Layers	Width	**Solver**	**Time (sec)**	**Backtracks**
Sliding	60	4	6	GRASP	746	32000
				SATO3	510	41800
				SBSAT	51	24000
Random	60	4	6	GRASP	>1000	-
				SATO3	85	11600
				SBSAT	.03	28

Table 1. Performance of SAT solvers on two LBFs

A variety of instances of LBFs were generated. To be used by existing SAT solvers, these were transformed to CNF in typical fashion: one clause for every *false* truth table row. Quine-McCluskey reductions were also performed but the best performing SAT solvers did not do any better on the resulting CNFs, generally, so such results are not reported here. Tables 1 and 2 are representative of what we observed. Generally, SBSAT did better than CNF solvers. Even if SBSAT needs to backtrack about the same number of times as another solver it still often does significantly better, probably because of the precomputed information stored in the state machines. But, it often needs to backtrack far less.

Solver	**Time (sec)**	**Backtracks**
GRASP	>1000	-
SATO3	22	6220
SBSAT	0.17	171

Table 2. Performance of SAT solvers on EXAMPLE

The results obtained so far are not conclusive. It is possible that some tweaking of SATO and GRASP would result in significantly better performance. In fact, we note that SBSAT's performance on EXAMPLE was due to optimally setting the parameter value of the "locally-skewed, globally-balanced" (LSGB) heuristic[3]. In addition, we have not tried the solver on enough representative inputs. However, we feel the results are sufficient to show the merit of continued development.

[3]This heuristic attempts to give high weight to search choices which bring inferences out of each SMURF sooner and yet approximately balance the search tree

3 Current Research and Development

The results to date demonstrate clearly that much more needs to be learned about optimizing SBSAT, both in stand-alone and distributed modes. The following points summarize our current research and development of the solver:

1. **Efficient use of lemmas**. Lemmas are used to explore the basis of a contradiction at the bottom of a backtracking tree search. With lemmas we compute the minimal basis, very quickly and efficiently, for the contradiction that was detected. In highly structured systems of expressions we find that the bases for these lemmas allow us to backtrack not to a parent node, but sometimes many nodes higher up the search DAG.

   Table 3 illustrates the amazing effectiveness of lemmas in pruning large search spaces. For the Static heuristic[4] the number of backtracks is cut by a factor of 80. But the gain in computational effort is only a factor of 22. In other words, we can maintain a rate of about 1640 backtracks per second without lemmas but this figure is reduced to only about 460 when backtracks are applied. Our goal for the backtrack rate without lemmas, based on the experience of others, is at least 20000 without lemmas and 10000 with lemmas. Clearly, we should be able to improve this situation considerably.

Heuristic	**Lemmas**	**Time (sec)**	**Backtracks**
Static	NO	376	617351
Static	YES	17	7847

Table 3. Performance of the solver on EXAMPLE with and without lemmas

   To meet this goal we are doing several things. Most straightforward is pick through the prototyped program and carefully replace sections with faster code. For example, we are reprogramming SBSAT using C instead of C++ and paying careful attention to the number of function calls, particularly in the brancher, and data structures.

   We are redesigning data structures which support lemma storage. Clearly, an essentially linear search, as is conducted in the prototype, is not adequate. Others have used more clever mechanisms such as a Trie.

   We are investigating the effects of node versus edge lemmas. A node lemma is a partial assignment which cannot lead to anything but a refutation, or no solution, if searching is continued beneath it. An edge lemma is the same except it applies beneath a literal on which a branch is taken. Two edge lemmas on opposite branches make a node lemma (by means of resolution where the branch variable is the pivot). So, the question of choosing node lemmas or edge lemmas or both to save is related to the question of which resolvents to save if resolution is applied. The results of Table 4 show this may not be a straightforward choice.

[4]A simple heuristic, used for reference, which never changes pre-computed variable weights during the search process.

We are considering a number of possible algorithms for choosing when, if ever, lemmas are to be discarded (too many lemmas appear to be too hard to manage). No one yet has an ideal rule for this and the answer may depend on an analysis of the given input.

Cache Type	Lemma Type	Cache Size		Lemmas		Time
			Backtracks	Hits	Grazes	(sec)
<none>	-	-	277675	-	-	92
MRH	edge	20	62350	13993	11561	34
	edge	40	28810	5162	12762	18
	edge	100	16084	2175	14281	13
	edge	∞	7540	794	11381	58
Boa	edge	∞	9169	910	12187	19
	node	∞	16899	3815	6957	22
	both	∞	8593	937	11039	22

Table 4. Effect of lemma cache size and replacement policy on searching

The results of some simple experiments on the effects of limiting the size of the lemma cache (newly added lemmas replace some lemma already existing in the cache if it is full) are shown in Table 4. In these experiments the Static heuristic was used on EXAMPLE. Under `Cache Type`, MRH means move the most recently hit lemmas to the "top" of the cache, and replace old lemmas from the "bottom" of the cache with newly generated ones. Boa means do not replace or move lemmas, only update literals appearing in the lemma from both ends[5] For MRH, `Cache Size` is the maximum number of lemmas allowed to be applicable at any one time. The column labeled `Hits` holds the number of times a lemma was applied to force a backtrack and the column `Grazes` holds the number of times a lemma was one variable away from being applied and, therefore, used to make a single variable inference.

2. **Use of autarkies**.

Informally, an autarky of a conjunction of expressions is a partial assignment that can reduce the problem of determining/finding a satisfying assignment for that conjunction to the problem of determining/finding a satisfying assignment for the conjunction of expressions which are not satisfied by the assignment. For example, a pure literal[6] in a CNF expression is autark with respect to any partial assignment and it can always be safely satisfied during search. With autarkies we can detect in highly structured problems that a piece of the expression space can be lopped off, thus avoiding unnecessary and costly backtracking. If the smaller system of independent expressions is satisifable then so is the larger system. If the smaller system of independent expressions is unsatisfiable then so is the larger set of expressions. Autarkies is a very exciting area a research that we are exploring. However, we are exploring autarkies over state machines, thus potentially making them more powerful than they are in the traditional CNF domain.

[5]Boa is short for Boa Constrictor and in this implementation a lemma is a doubly linked list of literals.
[6]A literal occurring either positively or negatively but not both in the expression.

3. **Efficient and effective heuristics**. We have implemented quite a few heuristics but have only just begun to explore the relationship between heuristics, lemmas, and inputs. Our results show how tricky heuristic design and use is. For illustration, Table 5 shows how effective, on a relatively highly structured input such as EXAMPLE, the Static heuristic is, in the absence of lemmas, compared to others such as the LSGB heuristic with default parameter 4.5. On the other hand, for less structured inputs such as the Sliding LBF mentioned earlier, the LSGB heuristic (parameter set to default of 4.5) appears to be quite superior to the others listed. A considerable effort will be needed to sort out this situation.

4. **Heuristic parameters**. The choice of parameter(s) for a given heuristic is equally tricky. This is illustrated in Table 5, where the LSGB heuristic is run with parameter set to 1.3 and 4.5, and Table 6, where the effect of various values of the single parameter K is given for the LSGB heuristic as applied to EXAMPLE. The LSGB heuristic selects a literal (that is, a branch) based on a weighting scheme over all active literals. In Table 6 we show what happens, for each K, when LSGB selected branch (normal branch denoted `N` in the Table under column `Dir.`) is taken and when the opposite branch (denoted `R` under column `Dir.` for reverse) is taken.

   Table 6 also shows the results of using lemmas and backjumps on EXAMPLE. A *backjump* can occur if the variable branched on does not show up in a refutation below that branch: in this case, taking the other branch will certainly lead to a refutation so it is not done. The number of backjumps taken is given in the column labeled `Jumps`. The column labeled `Both` holds the number of remaining backtracks (two branches each) made. The choice of parameter and its interaction with lemmas clearly needs to be studied further.

Input	**Heuristic**	**Lemmas**	**Time (sec)**	**Backtracks**
EXAMPLE	Dynamic	NO	1250	244020
	Static	NO	92	277675
	Future	NO	376	617351
	LSGB (4.5)	NO	270	369749
	LSGB (1.3)	NO	0.17	171
Sliding 60-4-6 LBF	Dynamic	NO	>86400	-
	Static	NO	601	2476152
	Future	NO	47	96536
	LSGB (4.5)	NO	6	9896
	LSGB (1.3)	NO	44	78565

Table 5. The solver applied to two LBFs with various heuristics and no lemmas

5. **Exposing inferences**. An important feature of each state machine is it exposes inferences early and quickly. When a variable is assigned a value, either by inference or choice, some machines change state. Associated with this change is a precomputed list of inferences (values to variables) which are implied. Inferences on this list are instantly broadcast to all objects needing the information since the time needed to

compute the inferences has already been charged to the preprocessing phase. Such inferences can often be broadcast before they would be in a corresponding CNF search, thereby reducing search effort. For example, consider f_1 to f_5 as follows:

v_1	v_3	v_8	f_1
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

v_3	v_7	v_9	f_2
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

v_4	v_6	v_{11}	f_3
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

v_5	v_9	v_{11}	f_4
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

v_5	v_8	v_{10}	f_5
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Setting v_1 to *true* causes f_1 to infer v_8 is *false* which causes f_5 to infer v_{10} is *true* which causes f_5 to infer v_5 is *true*, all before any choice point is reached. But there are many other ways to expose inferences faster. For example, consider f_1 to f_5 again as a complete formula: without loss, v_4 may be set to *true* and v_6 set to *false*, but then f_4 *can be inferred satisfied* because v_{11} appears in no unsatisfied function; doing so allows v_9 to be set to *false* and v_5, v_{10}, v_7 to be set to *true* followed by f_1 *inferred satisfied*, ending the search before a single choice point is reached! We expect to find and consider numerous additional, non-conventional ways to expose inferences.

6. **Analysis of a given input**. It is well known that an analysis of the given input, before applying a solver, can significantly improve the search effort. For example, there may be a total ordering of functions and of inputs such that, for all or most inputs, the functions using that input are located in a very narrow portion of the function ordering. If such an ordering is found, low searching effort can be guaranteed. As another example, significant portions of a given input may represent subinstances which are known for other reasons to be solvable efficiently. Such information can help reduce search effort significantly, particularly if some decomposition is possible as described in the next item. Finally, we mention that clustering of clausal inputs into function groups, possibly attempting to reconstruct relationships existing in a pretransformed formula, should help significantly with search. Inspection of a number of CNF formulas coming out of practical applications, particularly in formal verification, shows

numerous occurrences of the following pattern, among others:

$$(v_1 \vee v_2 \vee v_3 \vee \ldots \vee v_n) \wedge (\bar{v}_1 \vee \bar{v}_2) \wedge (\bar{v}_1 \vee \bar{v}_3) \wedge \ldots \wedge (\bar{v}_1 \vee \bar{v}_n)$$

which means v_1 is equivalent to the disjunction of $v_2, \ldots, v_n$. These $n+1$ clauses probably are the result of "flattening" to CNF. Recombining them to a state machine of n variables is easy to do and probably results in improved performance over the alternative which is to construct $n+1$ state machines, one for each clause.

		Backtracks		**Lemmas**		
K	**Dir.**	**Both**	**Jumps**	**Hits**	**Grazes**	**Time**
1.1	N	670	13	40	439	0.69
	R	2936	154	174	2681	4.32
1.3	N	142	0	3	21	0.15
	R	860	3	16	329	0.93
1.5	N	2785	2	384	5507	5.24
	R	457	2	13	283	0.52
2	N	11861	177	2265	26834	61.87
	R	4336	9	596	7856	10.54
2.5	N	13542	127	2837	34321	61.11
	R	3051	13	352	5853	5.74
3	N	10358	101	1301	19101	42.03
	R	17768	790	3698	41490	106.9
3.5	N	7656	104	1069	13019	26.68
	R	13016	706	2197	27002	66.07
4	N	7825	290	1222	14421	25.54
	R	8303	743	1353	15336	29.34
4.5	N	6597	261	1171	12159	19.4
	R	8170	514	1332	15578	29.27
5	N	5972	301	800	12290	17.76
	R	12428	904	1836	23949	70.54
6	N	20108	1121	3135	39386	167.19
	R	11393	1084	918	20751	65.53
7	N	19862	1090	3107	40886	149.82
	R	10268	912	1017	18208	50.01
8	N	17072	794	2590	36523	103.4
	R	8158	545	877	14739	29.81
9	N	16779	815	2626	35787	101.37
	R	7876	535	859	14247	28.98
10	N	16940	829	2625	35987	103.08
	R	7987	541	828	14333	29.12

Table 6. The solver using LSGB Heuristic on EXAMPLE: varying parameter K

We have so far done little to exploit these ideas, yet our direction and previous observations and results by others suggests we must.

7. **Decomposition of a given input**. One of the most understudied areas of great potential applicability to SAT solving aims to deal with the question of decomposing formulas into a few large orthogonal pieces. Then each piece may be solved independently and a solution to the given formula may be composed from non-interacting solutions to each piece. Variations on this idea relax somewhat the requirement of orthogonality. As an example, consider linear autarkies for CNF formulas. For a particular CNF formula ϕ, define a $\{0, \pm 1\}$ matrix A, with rows indexed on clauses of ϕ, columns indexed on variables of ϕ, and such that element $A_{i,j}$ is 1/-1/0 if clause i contains literal v_j/contains literal $\bar{v}_j$/does not contain variable v_j, respectively. Then a ± 1 vector $\mathbf{x}$ satisfying $A\mathbf{x} > \mathbf{b}$ is a solution to ϕ, where b_i is 2 minus the number of literals in clause i. A vector $\mathbf{x}$ of reals satisfying $A\mathbf{x} > 0$ is called a linear autarky of A. Linear autarkies generalize considerably the well known concept of *pure literal.* Let $\mathbf{x}$ be a linear autarky of A and rearrange the order of variables so that $\mathbf{x} = \{\mathbf{x}_1 | \mathbf{x}_2\}$ where all entries of $\mathbf{x}_2$ are 0 and all of $\mathbf{x}_1$ are non-zero. Round the elements of $\mathbf{x}_1$ to either -1 or 1, whichever is closest. Let ϕ_1 be ϕ with all clauses satisfied by the partial assignment of $\mathbf{x}_1$ removed. Then a solution to ϕ_1 is a solution to ϕ and if ϕ_1 has no solution then neither does ϕ. In this way, linear autarkies induce decompositions on CNF inputs. The following is known about linear autarkies for CNF formulas:

   (a) They can be found efficiently.

   (b) A unique autarky-free subformula can be found efficiently.

   (c) A number of polynomial time solvable classes of SAT are subsumed by a class based on linear autarkies.

   Theoretical results in this area are quite recent and no empirical results have been reported for CNF formulas. Nothing has been done for inputs such as the ones we work with. Hence, this area needs more study. Linear autarkies are only one of a number of decomposition methods[7] which may be applied efficiently and subsume known significant classes of SAT, but whose potential has not been fully studied. We propose to determine the effectiveness, empirically, of such ideas.

[7]See "Effective Logic Computation," by K. Truemper, Wiley, 1998 for some.

A Layered Boolean Functions

Let $\phi_L = \{f_{1,1}, \ldots, f_{1,m_1}, f_{2,1}, \ldots, f_{2,m_2}, f_{l-1,1}, \ldots, f_{l-1,m_{l-1}}, f_{l,1}, \ldots, f_{l,m_l}\}$ be a set of Boolean functions such that $f_{i,j}$, $1 \leq j \leq m_i$, depends only on the input vector if $i = 1$ or on the outputs of functions $f_{x,y}$, where $1 \leq x < i$, and the input vector if $i > 1$ (we let $m_l = m$, the number of "visible" outputs: that is the output vector consists of $\langle f_{l,1}, \ldots, f_{l,m}\rangle$). Then ϕ_L is said to be *layered* and has l layers. We show here that ϕ_L, $l > 2$, can be simulated without loss by a set of functions with $l = 2$.

Construct a two layer F from an l-layer ϕ_L as follows. Create $n + m_1 + m_2 + ... + m_{l-1}$ inputs and $m_2 + m_3 + ... + m_{l-1} + m$ outputs. Label all the inputs

$$p_{0,1}, \ldots, p_{0,n}, p_{1,1}, \ldots, p_{1,m_1}, \ldots, p_{l-1,1}, \ldots, p_{l-1,m_{l-1}}$$

consecutively, from left to right. The first n of these inputs correspond to the actual independent inputs of ϕ_L. The rest correspond to the "internal" interconnect points of ϕ_L. Label all the outputs

$$f'_{1,1}, \ldots, f'_{1,m_1}, f'_{2,1}, \ldots, f'_{2,m_2}, \ldots, f'_{l-1,m_{l-1}}, f_{l,1}, \ldots, f_{l,m}.$$

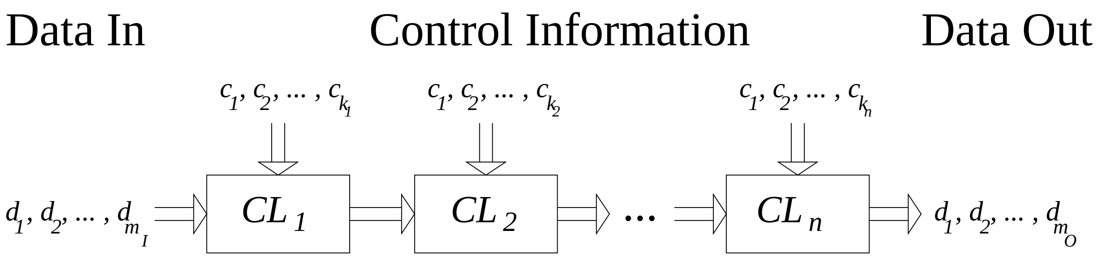
corresponding to the outputs of functions $f_{i,j}$. The truth table for $f'_{i,j}$ is obtained from the truth table for $f_{i,j}$ as follows. Let $xxx\ldots xx$ be an input pattern mapping $f_{i,j}$ to 1. Then, add the input $p_{i,j}$ to the right end (the truth table for $f'_{i,j}$ now has a new column on the right representing values of $p_{i,j}$) and have $f'_{i,j}$ map to 1 on input $xxx\ldots xx1$ and map to 0 on input $xxx\ldots xx0$. Let $yyy\ldots yy$ be an input pattern mapping $f_{i,j}$ to 0. Then, add $p_{i,j}$ to the right end and have $f'_{i,j}$ map to 0 on input $yyy\ldots yy1$ and map to 1 on input $yyy\ldots yy0$.

Now, consider the problem of determining an input pattern which causes a specific output vector t_o for a l-layer ϕ_L. In the two layer simulation the problem is to determine a pattern for $p_{0,1}, \ldots, p_{l-1,m_{l-1}}$ which causes $f_{l,1}, \ldots, f_{l,m}$ to have output t_o provided that $f'_{1,1}, \ldots, f'_{l-1,m_{l-1}}$ all have value 1. The solution to the l-layer problem and the two-layer simulation are identical.

B A Class of Examples

This is a description of a class of propositional problems inspired by the following integrated circuit problem: Does control information exist to map inputs DI to DO through n chunks of combinational logic?

For example, if DI and DO represent an unsecure/illegal state transition, then the propositional problem is to verify that the unsecure/illegal state is unreachable or to give a counter example of how the unsecure/illegal state can be reached from the initial state input.


Data In
Control Information
Data Out
$c_1, c_2, \ldots, c_{k_1}$
$c_1, c_2, \ldots, c_{k_2}$
$c_1, c_2, \ldots, c_{k_n}$
$d_1, d_2, \ldots, d_{m_I}$
CL_1
CL_2
...
CL_n
$d_1, d_2, \ldots, d_{m_O}$

Thus, as shown in the diagram above, we have m_I-bits of data-in and k_1-bits of control information going into some combinational logic CL_1, this output going into combinational logic CL_2 with k_2-bits of control information, ... , output from combinational logic CL_{n-1} into combinational logic CL_n with k_n-bits of control information to produce m_O-bits of data-out.