A Proof Engine Approach to Solving Combinational Design Automation Problems

Gunnar Andersson, Per Bjesse, Byron Cook
Prover Technology
{guan,bjesse,byron}@prover.com

Ziyad Hanna
Intel Corporation
ziyad.hanna@intel.com

ABSTRACT

There are many approaches available for solving combinational design automation problems encoded as tautology or satisfiability checks. Unfortunately there exists no single analysis that gives adequate performance for all problems of interest, and it is therefore critical to be able to combine approaches.

In this paper, we present a proof engine framework where individual analyses are viewed as strategies—functions between different proof states. By defining our proof engine in such a way that we can compose strategies to form new, more powerful, strategies we achieve synergistic effects between the individual methods. The resulting framework has enabled us to develop a small set of powerful composite default strategies.

We describe several strategies and their interplay; one of the strategies, variable instantiation, is new. The strength of our approach is demonstrated with experimental results showing that our default strategies can achieve up to several magnitudes of speed-up compared to BDD-based techniques and search-based satisfiability solvers such as zCHAFF.

Categories and Subject Descriptors

B.6.3 [**Hardware**]: Logic Design—*Design Aids*

General Terms

Algorithms, Design, Verification

1. INTRODUCTION

As many combinational design automation problems are NP- or coNP-hard, one fundamental approach to their solution is to model the problems as propositional logic formulas, and to apply proof methods to decide whether the formulas are tautologies or not.

DAC 2002, June 10-14, 2002, New Orleans, Louisiana, USA.

However, there exists a plethora of techniques for discharging propositional proof conditions, including (but not limited to) the DPLL method [5], Stålmarck's method [11], Binary Decision Diagrams [2], and rewriting. Each proof technique has a particular characteristic in terms of space and time behavior, sensitivity to the size of the system, and the particular domain it will work the best for. As a result, it is not unusual to combine techniques in different ways when solving particular classes of combinational design automation problems.

At Prover Technology, we develop and maintain a plug-in proof engine that is used to solve combinational logic problems in design automation tools. This proof engine, PROVER CL, is delivered with default composite analyses for a number of domains.

As developers and maintainers of PROVER CL, we need to experiment with new variations of the default analyses so that we can keep them updated. We also need to develop new analyses that work well in new problem domains, and on problems with particular characteristics. It is thus important that our framework is constructed in such a way that (1) we can get the maximum amount of synergy between the individual methods, and (2) we can construct composite analyses with a minimum amount of effort.

In this paper, we present an approach to addressing these issues that we have been developing for several years—initial results were reported in early 2000 [4].

Our approach builds on two key ideas. First of all, we structure our proof engine in such a way that we view individual proof techniques as *strategies*; that is, functions between different proof states. Second, we define our strategies to be compositional in the sense that they can be combined in any order to form more powerful proof-search strategies.

As our experimental results indicate, the resulting compositional approach allows us to get synergy effects that provide order of magnitude speedups compared to individual analyses such as the zCHAFF [8] search method, and state-of-the-art BDD-based analyses.

2. RELATED WORK

What we describe in this paper is an industrially proven framework for creating composite analyses for arbitrary combinational design automation problems (as opposed to just test pattern generation, for example). Our framework is carefully structured in such a way that any analysis from a

large set can be composed in any order. We are not aware of any previous attempt to come up with such a general framework for solving problems in all of the many subdomains of combinational design automation.

However, a number of previous approaches to solving problems in particular subdomains of combinational design automation can be viewed as composite strategies in the sense that we will present. This is especially true in the domain of combinational equivalence checking, where combinations of BDDs, SAT and rewriting have been suggested by a number of different researchers [3, 6, 7, 9]. The most compelling example is probably the filter-based analysis for combinational equivalence checking that was proposed by Jain and coworkers in [7]. In this paper, the overall analysis is a sequential composition of a small set of *filters*. Each filter inspects a set of conjectured equivalences, and selects a few of them that it tries to prove using some technique. The filter-based analysis is hence a particular simple composition of individual analyses, each of which we can see as a strategy.

The view of strategies that we adopt in this paper is similar to the notion of tactics and tacticals [10] developed for LCF-style theorem provers in the early 1970s. The ideas behind tactics and tacticals have since proved to be fruitful enough that most theorem provers for first order and higher-order logics use variations on this approach.

In essence, our contributions in this paper are (1) to take the research from the higher-order domains and apply it in the domain of completely automated correctness proofs for combinational logic, (2) to describe how some classical proof methods and a novel method for simplifying SAT problems can be viewed as strategies, and (3) to present our experiences and some experimental results that illustrate the power of the approach.

3. PRIMITIVE STRATEGIES

The core problem that we are interested in solving is as follows. Given a propositional logic formula ϕ, we want to decide whether this formula is a *tautology*—that is, we want to decide whether the formula evaluates to true for all assignments to the variables in ϕ. We solve this problem by trying to prove ϕ using a *reductio ad absurdum* argument, where we use a combination of different analyses to check whether the negation of the formula is unsatisfiable.

We model individual analyses as strategies; that is, as functions

$$f : (\texttt{PState} \times \texttt{Parameters}) \rightarrow ([\texttt{PState}] \times \texttt{Res})$$

that take a proof state and some parameters, and return a list of new proof states together with information about the result of the analysis. The result is one of the values `Sat`, `Unsat`, and `Indet`, and we require that all strategies return `Indet` precisely when the returned list of proof states is nonempty.

A proof state contains a number of objectives $\texttt{Obj}_1 \ldots \texttt{Obj}_k$, and a set of current-time assumptions `Asmpt`. The objectives are formulas, and the assumptions are equivalence classes of negated and unnegated subformulas of the objectives.

Intuitively, a proof state represents the problem of deciding whether there is an assignment to the variables in the objectives that (1) makes all the formulas in each assumption equivalence class evaluate to the same value, and that (2) makes all of the objectives evaluate to true. We model the situation at the start of the overall proof search as a proof state that contains no assumptions together with the single objective $\neg\phi$; this represents the notion that we want to prove that ϕ is a tautology by deciding whether there is a model of the negation of ϕ.

Now, assume that *AllSat* is a mathematical function that decides problems represented as proof states. A strategy is *correct* if it fulfills two criterias:

1. If the list of returned proof states l is empty, then the returned outcome agrees with the result of $\mathit{AllSat}(p)$.
2. If the list of returned proof states l is nonempty, then $\mathit{AllSat}(p)$ holds precisely when $\mathit{AllSat}(p_2)$ holds for some proof state p_2 in l.

We require that all strategies in our framework are correct.

There are a number of additional properties of strategies that we can define as follows. Assume that the strategy *strat* is given the proof state p and returns the list of new proof states l together with the evaluation result res. The call to the strategy *strat* is

- *decisive* if l is an empty list.
- *refining* if l is a singleton list containing the original proof state with some extra assumptions.
- *splitting* if l is a list of length at least 2.

Most strategies are decisive in some cases, refining and splitting in others. However, if we want to guarantee that we can solve our top level goal, we need to construct an overall strategy that always is decisive.

Let us now consider how three previously known and one novel algorithm for satisfiability solving can be cast as strategies in our framework.

3.1 Stålmarck's saturation method

Stålmarck's *saturation method* is an algorithm for propositional proof [11] that has been applied to solve a variety of problems in software and hardware verification.

An application of the saturation algorithm takes three parameters as input: (1) a set of formulas $\phi_1, \ldots, \phi_n$, (2) an equivalence relation over negated and unnegated subformulas of the set of formulas, and (3) a number k, called the *degree of saturation*. Given these inputs, the saturation algorithm generates an extension of the equivalence relation that provably must hold if the original equivalence relation holds. The key idea in the saturation algorithm is to extend the relation with new equivalences that can be inferred from the original relation using *short* proofs. A proof is k-short if not more than k simultaneous assumptions are needed in the Dilemma [11] proof system.

Saturation is normally used for generating propositional proofs by contradiction: We first construct an equivalence relation where the negation of the formula is put in the same class as the value *True*, and then apply the saturation algorithm. If the formula is provable, there will exist a degree of saturation for which the resulting equivalence relation is *contradictory*, in the sense that the values *True* and *False* are in the same equivalence class.

In the present context, we are interested in constructing a strategy that we can use to process problems represented as proof states. The approach we take is to construct a wrapper around the saturation procedure that takes a proof state as argument together with a saturation degree. From

the proof state, we construct an initial equivalence relation from the assumptions `Asmpt`, and add the formula $\texttt{Obj}_1 \wedge \ldots \wedge \texttt{Obj}_k$ to the class containing *True*. We then apply the saturation algorithm to the equivalence relation, with the given degree of saturation, and compute the strategy result in the following way: If the result of the saturation is contradictory, we return the result $([\,], \texttt{Unsat})$. If the equivalence relation is noncontradictory, and all variables in the formula are in the equivalence classes containing *True* and *False*, then we return the result $([\,], \texttt{Sat})$. Otherwise we return $([\texttt{PState}'], \texttt{Indet})$, where $\texttt{PState}'$ is the original proof state augmented with assumptions corresponding to the resulting equivalence classes. Our saturation strategy is thus either decisive or refining.

3.2 DPLL

The Davis-Putnam-Loveland-Logemann algorithm [5] is a popular search-based satisfiability method. The DPLL procedure takes as input a formula in Conjunctive Normal Form (CNF). A CNF formula is a conjunction $\bigwedge_{k=1}^{n} C_k$, where each *clause* C_k is a disjunction $\bigvee_{i=1}^{m_k} l_i$ of literals—negated or unnegated variables.

The DPLL procedure attempts to find a satisfying valuation for the variables in the CNF formula by recursively selecting a variable x and case-splitting on its value. If either of the problem instances where x is set to true or false is satisfiable, then the whole problem must be satisfiable. Equivalently, if none of the instances are satisfiable, then the whole problem is unsatisfiable.

A naive implementation of the DPLL procedure that only uses case splitting to search for a satisfying assignment is theoretically both sound and complete. However, it is rarely computationally feasible to apply it to real life formulas as too many case splits will be required to decide satisfiability. As a consequence, most DPLL implementations use a number of optimizations.

One of the most important optimizations is the generation of *conflict clauses*. Whenever it is detected that the currently explored partial assignment makes some clause evaluate to false, it is possible to find a subset of the partial assignment which is the reason for this conflict. Assume for example that this subset is the valuation $(x := 0, y := 1, z := 0)$. If the set of clauses at hand is satisfiable, we know that none of its models can contain this particular subvaluation. We can express this piece of information as a constraint—a conflict clause—of the form $x \vee \neg y \vee z$. Now, as this clause is a consequence of the satisfiability of the CNF formula, it is sound to add it to our set of clauses, where it will restrict the remainder of the search.

Let us now consider how to make a strategy out of the DPLL procedure. Our strategy takes a proof state and a timeout value as arguments, encodes the proof state in clausal form, and runs our DPLL implementation on the resulting CNF formula. If the algorithm terminates without timing out, then we can decide the problem at hand. However, even if our search times out, valuable information will have been generated in the form of conflict clauses that we can use to augment the proof state we return. There are a number of choices for how we can do that. Here, we go with the simplest possible solution, and investigate the generated conflict clauses to see if we have any clauses containing a single literal. If so, then we add a new assumption for each one literal clause in such a way that the variable is forced to the value indicated by the literal. Just as in the case of the saturation strategy, the DPLL strategy is hence either decisive or refining.

3.3 BDD-based cut

A standard approach to deciding the satisfiability of a propositional formula is to construct its Binary Decision Diagram (BDD) and check whether there is at least one path from the BDD's top node to the constant node *True*. However, it is often not feasible to build BDDs for complex formulas. We therefore adopt the ideas in [1] and construct our BDD-based strategy in such a way that the user can set a limit on the size of the formula that we build a BDD for.

We define the *construction depth* of a node in a parse tree to be the number of nodes between it and the root of the tree. Hence, the root of a parse tree has construction depth 0. Our BDD strategy takes a proof state as argument together with a construction depth k, and makes a cut in the parse tree for $\texttt{Obj}_1 \wedge \ldots \wedge \texttt{Obj}_k$ at depth k. Assume that this cut intersects the nodes $\texttt{Node}_1 \ldots \texttt{Node}_k$. We substitute fresh variables $\texttt{Var}_1 \ldots \texttt{Var}_k$ for each of the intersecting nodes, and build a BDD for the top node of the resulting simplified parse tree.

If the cut depth of the parse tree is equal to or greater than the maximum construction depth of a variable in the original formula, then the resulting BDD allows us to decide the problem represented by our proof state. However, if the parse tree is cut at a lower level, what do we know?

Consider the paths in the generated BDD that end up at the *True* terminal. Each such path represents a valuation of a subset of the original variables and internal nodes that guarantees that the top node becomes true. Note that such a path may or may not correspond to a satisfying assignment: Given a set of values for internal nodes in the formula, we can not be sure that there exists an extension of the assignment to the original variables that generates this particular combination of internal values. However, we do know that if there exists a satisfying assignment, then it will be covered by one of the paths to *True*. We therefore define our BDD strategy to be splitting when it is not decisive by letting it generate one derived subproblem for each path to *True* in the following way.

Assume that we have a path to *True* that encodes the assignment $(\texttt{Var}_1 := 0, \texttt{Var}_2 := 1, \texttt{Var}_3 := 0)$, and that $\mathit{Def}(x)$ is an operator that returns the formula that is assigned the name x. Then the new proof state that we generate to represent the derived subproblem contains the objectives $\neg\, \mathit{Def}(\texttt{Var}_1)$, $\mathit{Def}(\texttt{Var}_2)$ and $\neg\, \mathit{Def}(\texttt{Var}_3)$, together with the previous assumptions.

3.4 Variable instantiation

The previous strategies are all based on previously known algorithms. We now present *variable instantiation*, one of the many in-house strategies that we have developed.

Let us define two formulas to be *equisatisfiable* if one is satisfiable precisely if the other is satisfiable. Now, assume that we want to check the satisfiability of a formula ϕ that contains the variable x. Also assume that we know that ϕ is equisatisfiable with the result of substituting the value *val* for every occurrence of x in ϕ. Then it clearly suffices to check the instantiated, simpler, formula for which there are only half as many assignments possible.

We can decide whether it is safe to instantiate a given vari-

able as follows. Consider the case where we want to know whether it is safe to instantiate x in ϕ with the value *True*. If $\phi(x{:=}\mathit{True})$ is satisfiable, it is clear that ϕ is satisfiable. However, in order to safely be able to set x to *True*, we also need to know that ϕ is unsatisfiable when $\phi(x := \mathit{True})$ is unsatisfiable. We can check whether this holds by trying to prove the formula $\phi(x{:=}\mathit{False}) \rightarrow \phi(x{:=}\mathit{True})$: The provability of this formula implies that if $\phi(x{:=}\mathit{True})$ is unsatisfiable, then there can be no models at all for ϕ, regardless of the value of x.

Let us now consider how to construct a strategy based on variable instantiation. Our variable instantiation strategy takes a proof state containing a set of objectives $\texttt{Obj}_1 \ldots \texttt{Obj}_k$ as input, and a set of assumptions `Asmpt`. For each variable that is not in the equivalence classes of *True* and *False* we apply variable instantiation in the following way.

Assume that the current variable under consideration is x. To check whether x can be set to *True*, we construct a new proof state containing the original assumptions, and the single objective

$$\neg(\bigwedge_{i=0}^{k} \texttt{Obj}_i(x{:=}\mathit{False}) \;\rightarrow\; \bigwedge_{i=0}^{k} \texttt{Obj}_i(x{:=}\mathit{True}))$$

We then invoke some fast but potentially incomplete strategy, say saturation of degree 0, and check the result. If the result is `Unsat`, then it is safe to set x to *True*. If not, we check whether it is safe to set x to *False* in the same way. Once we know that we can set a variable to some value, we substitute this value in all the original objectives and assumptions and move on to the next variable. If there are no variables left after an iteration over the variables, then we have solved the problem. Otherwise we return the resulting proof state together with the result `Indet`. The variable instantiation strategy is thus decisive or refining.

4. COMPOSING STRATEGIES

The primitive strategies in Section 3 are all carefully designed to behave in a similar fashion: They take a parameter as argument that directly or indirectly controls how much time will be spent on the problem, and then apply some analysis to the given proof state. This analysis either decides the problem given the resource bounds, or generates one or more simpler problems.

However, so far we have no way to compose the individual analyses to form new, more powerful, analyses. We now explore this.

Recall our definition of individual strategies as functions

$$f : (\texttt{PState} \times \texttt{Parameters}) \rightarrow ([\texttt{PState}] \times \texttt{Res})$$

that takes a proof state and some parameters `Parameters` as inputs. In the case of the saturation strategy, the single parameter is the saturation degree, and in the case of the DPLL strategy the parameter is the cutoff time. Informally, the way we will do composition is to allow `Parameters` to also contain an analysis to use if the problem is indeterminate given the resource constraints. After the post-processing analysis has been applied to every generated subproblem, the primitive strategies computes the overall result as follows. If one of the subproblems is diagnosed as satisfiable by the post-processing analysis, then the overall problem is satisfiable. If all are contradictory, then the overall problem is contradictory. Otherwise the result is indeterminate.

In order for every strategy to be able to use any other strategy to post-process indeterminate results, we need to settle on a standardized strategy interface. We define the type `StratI` of *strategy instances* to be the type of functions of the form

$$f_{inst} : \texttt{PState} \rightarrow ([\texttt{PState}] \times \texttt{Res})$$

and provide a trivial strategy instance `triv` that takes a proof state ps and returns $([ps], \texttt{Indet})$. We can now modify the signature of our example primitive strategies to take a post-processing strategy instance as an additional argument:

$$\texttt{sat} : (\texttt{PState} \times \texttt{Degree} \times \texttt{StratI}) \rightarrow ([\texttt{PState}] \times \texttt{Res})$$
$$\texttt{cut} : (\texttt{PState} \times \texttt{Level} \times \texttt{StratI}) \rightarrow ([\texttt{PState}] \times \texttt{Res})$$
$$\texttt{dpll} : (\texttt{PState} \times \texttt{Time} \times \texttt{StratI}) \rightarrow ([\texttt{PState}] \times \texttt{Res})$$
$$\texttt{inst} : (\texttt{PState} \times \texttt{StratI} \times \texttt{StratI}) \rightarrow ([\texttt{PState}] \times \texttt{Res})$$

In the case of the variable instantiation strategy, this results in two strategy instance arguments. The first is used to decide instantiation safety, and the second is used for post-processing.

As things stand, the only strategy instance we can provide for post-processing is the trivial strategy instance `triv`, and this is not very interesting. However, it is easy to turn a strategy into a strategy instance by wrapping it in a function that provides particular parameter values (including the strategy instance that is to be used for the proof states resulting from an indeterminate result). We can hence achieve sequential composition by using appropriate wrappers and `triv` to chain together individual strategies from the bottom up.

As an example, assume that we are interested in writing a composite strategy instance that applies DPLL for a hundred seconds only. We can do this by writing a wrapper that passes `dpll` the given proof state together with the value 100 and the strategy instance `triv`. After the DPLL invocation has finished, the wrapper takes the result from the DPLL strategy, and returns it. The resulting composite strategy instance can then be used to build a new strategy instance that first does a BDD cut on construction depth 20, and then applies DPLL to the resulting problems for a hundred seconds:

```
dpllI(ProofState) :=
  return dpll (ProofState, 100, triv)

cutDpll(ProofState) :=
  return cut (ProofState, 20, dpllI)
```

Note that we are not restricted to constructing strategy instances that are *static* in the sense that they always execute a set of strategies in a particular order, without any flexibility. We are writing the wrappers ourselves, so it is simple to construct strategy instances that examines the result returned from a strategy and acts based on the outcome. For example, assume that we want to use a BDD cut to generate a list of new problems, and apply `dpll` for some period of time on the results that fulfill some predicate, and saturation on the others. The flexibility of our framework makes this easy to do by providing an appropriate instance wrapper in the call to `cut`.

In our framework, we can thus construct complex composite analyses with a few lines of programming. However,

Instance	PROVER CL (s)	ZCHAFF (s)
c1355	0.3	2.4
c1908_bug	0.0	3.2
c1908	0.0	3.5
c2670_bug	0.2	0.0
c2670	0.1	1.8
c3540_bug	1.0	0.0
c3540	1.0	51.6
c432	0.0	0.1
c499	0.1	4.5
c5315_bug	1.7	1.1
c5315	0.8	46.4
c6288	4.2	[>60 min]
c7552_bug	2.2	0.4
c7552	2.2	84.5
c880	0.1	1.4

Table 1: ISCAS'85 benchmark results

when we are experimenting with different compositions (for example, when developing default analyses), it is desirable to avoid writing code and recompiling altogether. To address this issue, we have developed a domain specific language for describing composite strategy instances, which we interpret using a command line tool called CAPTAIN PROVE. This tool takes a formula as an argument, together with a text file that describes a strategy instance that it should apply to decide provability of the formula. Internally, CAPTAIN PROVE constructs an initial proof state, and then applies the described instance.

A simple description of a composite analysis in our strategy language looks as follows:

```
cut  10.
instantiate (dpll 10.).
sat  1.
dpll 100.
```

This strategy first does a cut at level 10. The resulting proof states are then refined by instantiation, where DPLL for ten seconds is used to decide whether a variable can be instantiated. Next, saturation of degree one is performed, followed by a hundred seconds of DPLL.

5. EXPERIMENTAL RESULTS

In order to demonstrate the power of the strategy approach, we have run three sets of experiments. Our objective is to demonstrate that the synergy effects between the many analyses that we combine can generate order of magnitude speedups, compared to individual techniques.

In the first two sets of experiments, we solve problems from the public benchmark collections ISCAS'85 and Satlib. The two suites represent different classes of design automation problems: The ISCAS'85 suite contains combinational equivalence checking problems, and the Satlib suite consists of bounded model checking problems. In the experiments, we compare the result from PROVER CL's default analyses against results from ZCHAFF [8], a state-of-the-art DPLL solver. The version of PROVER CL we are using here is version 4.1α10. All measurements for the public suites are done on a 1.5 GHz P4 with 2 GB of memory, running Red Hat Linux 7.1.

Instance	PROVER CL (s)	ZCHAFF (s)
ibm-1	0.5	2.5
ibm-2	0.0	0.0
ibm-3	0.8	0.9
ibm-4	1.2	4.3
ibm-5	0.2	0.4
ibm-6	5.2	14.6
ibm-7	0.2	0.2
galileo-8	7.0	92.3
galileo-9	8.6	140.2
ibm-10	7.1	404.2
ibm-11	3.5	34.8
ibm-12	16.4	165.4
ibm-13	7.4	10.1

Table 2: SATLIB benchmark results

In the third set of experiments we present results that have been generated in-house at Intel. Here we compare the performance of PROVER CL version 4.1α9, ZCHAFF, and a BDD-based tool developed internally at Intel. The measurements for this suite were done on a 1 GHz PIII with 1 GB of memory, running Red Hat Linux 6.2.

A note on strategies used in the experiments: For the ISCAS'85 and Intel experiments, we use the default strategy for combinational equivalence checking. For the Satlib experiments, we use the default strategy for bounded model checking. These default strategies were not devised to work well on these particular examples, and no changes to the default parameters were made in any of the experiments.

ISCAS'85

The ISCAS'85 suite consists of 15 combinational equivalence checking problems. Here, the objective is to show that all output signals from a pair of circuits are equivalent. One of the circuits in the pair is the original ISCAS'85 circuit and the other circuit is an optimized version. Due to erroneous optimizations, it may be the case that the optimized circuit is not equivalent to the original circuit.

We have run the experiments as follows. Using PROVER CL, we verify that individual pairs of outsignals from the circuits are equivalent. The time given for each circuit is the total time for deciding equivalence for every single outsignal. For ZCHAFF we use the CNF files produced by João Marques da Silva. Each CNF file corresponds to the problem of deciding whether the conjunction of individual equivalences for the outsignals is true or not. Note that this is an easier problem than that solved by PROVER CL.

We present the results in Table 1. As can be seen, it is hard to say anything about the relative performance of the engines for trivial examples that take less than a few seconds. However, for problems that take more than a couple of seconds, the trend is uniform: PROVER CL is one or more orders of magnitudes faster than ZCHAFF. Moreover, not only is PROVER CL always the faster engine, but the distribution of runtimes is also more uniform than for ZCHAFF.

Satlib

The Satlib benchmark suite contains 13 industrial model checking problems from IBM and Galileo (a telecommunication hardware manufacturer). Here, the objective is to decide whether there exists a run of a particular length from

Problem	PROVER CL (s)	ZCHAFF (s)	BDDs (s)
1	0.1	18.3	[$> 10^7$ nodes]
2	0.1	0.8	[$> 10^7$ nodes]
3	0.2	222.6	[$> 10^7$ nodes]
4	0.9	49.9	[$> 10^7$ nodes]
5	0.1	9.5	[$> 10^7$ nodes]
6	0.1	0.3	[$> 10^7$ nodes]
7	0.8	[>60 min]	[$> 10^7$ nodes]
8	9.9	101.5	[$> 10^7$ nodes]
9	0.3	15.5	[$> 10^7$ nodes]
10	450.2	[>60 min]	110.0
11	2.8	[>60 min]	[$> 10^7$ nodes]
12	139	232.5	[$> 10^7$ nodes]

Table 3: Intel benchmark results

the initial states of a circuit that ends up in a state violating a design invariant. For all of the problems, there exists such a run.

We have run the experiments as follows. The Satlib benchmarks are only available as pregenerated conjunctive normal form formulas optimized for CNF solvers like ZCHAFF. Unfortunately, PROVER CL operates best on formulas in full propositional logic; the conversion to CNF destroys valuable structural information. Since our objective is to compare performance on design automation problems, this situation is very troublesome—we can only do a partial translation back to full propositional logic for PROVER CL (this translation takes negligible time). Nevertheless, as can be seen in Table 2, PROVER CL outperforms ZCHAFF on all examples.

Intel benchmarks

The ISCAS and Satlib benchmark suites are relatively easy both for ZCHAFF and PROVER CL. We now consider another set of benchmarks where this is not the case.

The Intel problem set contains twelve combinational equivalence checking problems. Here, the objective is again to check that an optimized and an unoptimized version of a net behave the same. In ten of the cases, there is a bug, and in two cases the two nets are equivalent.

The experiments have been run as follows: We compare the performance of PROVER CL, ZCHAFF, and a BDD-based method developed internally at Intel. The problem instances are generated in full propositional logic for PROVER CL, and in clausal form for ZCHAFF. In the experiments, a timeout of one hour is used for all methods. For BDDs, we also abort when ten million BDD-nodes have been built.

We present the results in Table 3, where problems 8 and 9 are the two equivalent pairs of circuits. As can be seen, the BDD-based method runs out of memory in all but a single case. However, for the example where it finishes, it is the fastest method. ZCHAFF is always slower than PROVER CL, and times out in three cases. PROVER CL completes all benchmarks (in all cases except two in less than ten seconds), and performs between one and several orders of magnitude better than ZCHAFF on examples that are not trivial for both methods.

6. CONCLUSIONS

In this paper, we have taken an idea that has proven fruitful in other domains—semi-interactive theorem proving in rich logics—and transferred it to the domain of propositional proof search, with the aim to speed up combinational design automation problems. We have introduced the underlying ideas, presented some particular strategies, one of which is new, and discussed how they are integrated in Prover Technology's combinational proof engine plug-in PROVER CL.

The end result of the research we have presented here provides three important benefits. First of all, many of the users of our technology need state-of-the-art performance without modifying parameters, and we provide this in our default strategies. Second, the customizability of our engine makes it possible for power users to construct analyses that are tuned for the particular set of in-house problems that need to be solved. Third, for many customers, it is vital that our tool makes it easy to expand the existing set of strategies with new analyses.

Ongoing and future work includes to add more primitive analyses to the current set of basic building blocks, as they become known. Up to now, every time we have expanded our set of primitive building blocks, we have been able to extend the range of our default strategies substantially. Moreover, we are continuing development of default analyses for new problem domains.

7. REFERENCES

[1] D. Brand. Verification of large synthesized designs. In *Proc. Int. Conf. on Computer Aided Verification*, 1993.

[2] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, Aug. 1986.

[3] J. R. Burch and V. Singhal. Tight integration of combinational verification methods. In *Proc. Int. Conf. on Computer Aided Verification*, 1998.

[4] K. Claessen and G. Stålmarck. A framework for propositional proof strategies. Technical report, Chalmers University of Technology, Computing Sciences Dept., January 2000.

[5] M. Davis, G. Logeman, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(394–397), 1962.

[6] S.-Y. Huang and K.-T. Cheng. *Formal Equivalence Checking and Design Debugging*. Kluwer Academic Publishers, 1998.

[7] J. Jain, R. Mukherjee, and K. Takayama. An efficient filter-based approach for combinational verification. *Fujitsu Scientific and Technical Journal*, 36(1):17–23, 2000.

[8] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. 39^{th} Design Automation Conference*, 2001.

[9] V. Paruthi and A. Kuehlmann. Equivalence checking using a structural SAT-solver, BDDs, and simulation. In *Proc. Int. Conf. on Computer Design*, 2000.

[10] L. Paulson. Tactics and tacticals in Cambridge LCF, 1979.

[11] G. Stålmarck. A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula. Swedish Patent No. 467 076 (approved 1992), US patent No. 5 276 897 (1994), European Patent No. 0403 454 (1995).