

unsatisfied clauses. We first identify all variables that are contained in the unsatisfied clauses, we will call these variables, b_i . We examine all clauses that contain variables b_i , and add the clause to F' only if there is no alternate variable not in b_i that satisfies the clause. In our example, we examine $f_4, f_5, f_6, f_7, f_8, f_9$ and f_{10} as possible clauses to add to F' . We only add f_7 and f_8 , for they are not satisfied by any other variables.

$$F' = (v'_5 + v_6)(v_2 + v'_6)(v'_2 + v_5)$$

We see that both clauses f_7 and f_8 contain v_2 . As a result of adding these clauses to F' , we must now consider all of the clauses that contain v_2 as possible clauses to be added to F' . In this case, we look at f_1, f_2 and f_6 . All of these clauses are satisfied by variables other than v_2, v_5, v_6 , and therefore we do not add any of the clauses to F' .

We can now solve F' over the variables v_2, v_5, v_6 . The size of our instance is decreased in size from ten clauses to three (even with the EC). Therefore, the additional overhead for solving the modified instance would be minimal.

Often, a single synthesis step is followed by a number of consecutive synthesis steps. Therefore, if we want to avoid numerous changes to all steps, we have to preserve as much as possible of the initial solution at the higher levels of abstraction.

The goal is to resolve a modified instance in such a way that we minimize the number of changes to the original solution. For example, assume that we have the following SAT instance, F , and satisfying truth assignment, S .

$$F = (v_1 + v_2 + v_4)(v_1 + v_4 + v'_5)(v'_1 + v'_3 + v_4) \\ (v_2 + v_3 + v_5)(v'_2 + v_4 + v_5)(v_3 + v'_4 + v_5) \\ S = \{v_1 = 1, v_2 = 1, v_3 = 0, v_4 = 0, v_5 = 1\}$$

If we add the clauses $(v'_2 + v_3 + v_4)(v_1 + v'_2 + v'_5)$ the formula F becomes unsatisfied. We now must resolve the instance. There are multiple satisfying assignments for F , for instance consider the following two assignments.

$$S_1 = \{v_1 = 0, v_2 = 1, v_3 = 1, v_4 = 1, v_5 = 0\} \\ S_2 = \{v_1 = 1, v_2 = 0, v_3 = 0, v_4 = 0, v_5 = 1\}$$

If we select the S_2 , we preserve four out of the five assignments, and therefore make the minimal amount of design changes. However, if we select the S_1 , only one of the five variables assignments is preserved, and as a result the design would need to be almost completely redesigned.

2. RELATED WORK

In the last two decades, combinatorial optimization in general, and integer and linear programming in particular, have attracted a great deal of attention. Numerous high quality books appeared including [10, 12]. Quality textbooks on integer programming include [11, 14]. ILP has been widely used in CAD for a great variety of optimization tasks, ranging from physical CAD [8] to behavioral synthesis[2].

It appears that one of the first efforts for EC was performed at Hitachi. Shinsha et al. [13] described an incremental logic synthesis technique for supporting changes in the physical design stage. They demonstrated the effectiveness of the approach on the Hitachi M68XH design. Later, many efforts were made at the logic synthesis level [4, 7].

In a sense, our work is closest to the work by Kirovski et al [5]. They defined the engineering change problem as constraint manipulation where the graph is restructured in such a way that it supports EC. However, there are numerous differences and novelties. First, for the first time we introduce the notion of preserving EC and the quantified notion of enabling EC. More importantly, the technique in [5] is restricted to graph coloring and scheduling, does

not guarantee optimality, and does not opt for successive application to new requests. Most importantly, their technique can only handle some specific changes in specification, whereas the new technique is completely general.

Boolean Satisfiability is among the most popular generic NP-complete problems with numerous applications both in CAD and other application domains. Excellent surveys on SAT include [3, 9].

3. PRELIMINARIES

In order to make the paper self-sufficient, we survey relevant background material about ILP methodology and its use, the terminologies that we use throughout the paper, along with the formal definition of engineering change.

There are several techniques how one can solve ILP or LP. For LP, the most widely used is the SIMPLEX approach that has an exponential runtime in the worst case. However, on an overwhelming number of practical instances, SIMPLEX is not just of polynomial complexity, but also very fast. During the last decade, a number of practical and powerful interior point algorithms have been successfully implemented. These algorithms have guaranteed polynomial runtimes.

ILP deals with problems where a function is to be maximized or minimized and the variables are constrained by inequality and equality constraints and/or integral restrictions. The objective function as well as the inequality or equality constraints are linear.

$$\max\{cx + hy : Ax + Gy \leq b, x \in Z_+^n, y \in \mathbb{R}_+^p\} \quad (1)$$

We define the mixed integer programming problem (MIP) as 1, where Z_+^n is the set of non-negative integral n -dimensional vectors, \mathbb{R}_+^p is the set of non-negative integral p -dimensional vectors, and $x = \{x_1, \dots, x_n\}$ and $y = \{y_1, \dots, y_p\}$ are the variables and unknowns of the problem.

We assume that all numbers are rational. For our purposes, we use a special case of the MIP where all variables are integer values. The integer linear programming (ILP) is defined as:

$$\max\{cx : Ax \leq b, x \in Z_+^n\} \quad (2)$$

More specifically, we restrict the integer values of x to be 0 or 1. This is called 0-1 ILP where x is redefined to be $x \in B^n$ where B^n is a set of n -dimensional binary vector.

For the sake of completeness and due to the fact that most often ILP are usually solved using relaxation to the LP problem, we conclude this section by stating the LP problem.

$$\max\{hy : Gy \leq b, y \in \mathbb{R}_+^p\} \quad (3)$$

A very important and often used special case of MIP is one where all variables are 0-1 variables that means that each variable represents a binary decision. Specifically, it is common to model a variety of problems using the dichotomy formulation for variables with respect to a specific event.

$$x = \begin{cases} 1, & \text{if the event occurs} \\ 0, & \text{otherwise.} \end{cases}$$

We define ILP using the following form.

$$Y = \text{MAX}(cx) \quad (4)$$

$$Ax \leq b \quad (5)$$

We define $x = \{x_1, \dots, x_n\}$ and $p = \{p_1, \dots, p_n\}$ as 0-1 variables. We denote the original or previous assignments of the instance by $P = (p_1, p_2, \dots, p_n)$ and the new assignments, after EC, by $X = (x_1, x_2, \dots, x_n)$.

Boolean Satisfiability is a NP-complete problem that is often used in CAD tasks. The problem can be formulated as follows[1].

Satisfiability

Instance: A set U of variables and a collection C of clauses over U .

Question: Is there a satisfying truth assignment for C ?

We formulate the SAT problem in the form of an ILP by using the ILP formulation of the set cover problem as an intermediate step. The set cover problem can be formally defined as follows:

Set Cover

Instance: Collection C of subsets of a finite set S , positive $K \leq |C|$.

Question: Does C contain a cover for S of size K or less, i.e., a subset C' of C with $|C'| \leq K$ such that every element of S belongs to at least one member of C' ?

The ILP formulation for the set cover problem can be specified in the following way.

$$x_i = \begin{cases} 1, & \text{if subset } C_i \text{ is selected} \\ 0, & \text{otherwise.} \end{cases}$$

$$A_{ij} = \begin{cases} 1, & \text{if element } S_j \text{ is covered by subset } C_i \\ 0, & \text{otherwise.} \end{cases}$$

We use the ILP formulation (4) and (5), where we define c as a negative identity vector and b as a positive identity vector. The objective function for the set cover problem is to minimize the number of subsets used to cover the set S . The constraints are that each element in S must be covered by at least one subset.

We now formulate the SAT problem using the set cover ILP formulation. We define each element in the finite set S as a single clause in the SAT formula. The collection of subsets C are the variables in both complemented and uncomplemented forms independently. Each subset C_i contains all clauses as its elements, i.e. elements of S , in which the variable i appears in. Therefore, we have twice as many subsets as there are variables in the instance, one subset to represent the complemented and one to represent the uncomplemented version of the variable. Because no variable can be assigned to both the uncomplemented form and complemented form at the same time, we add a constraint (6), one for each variable, where n is the number of variables in the SAT problem. We define the objective function and the other constraints in the same way as the set cover problem.

$$x_i + x_{i+n} \leq 1 \quad (6)$$

To illustrate the formulation, consider the following SAT instance, F , with three variables and three clauses.

$$F = (v_1 + v_2)(v_2 + v_3)(v_1 + v_3)$$

In this case, we define $x = \{x_1, \dots, x_6\}$, where x_1, \dots, x_3 represent uncomplemented versions of the variables, and x_4, \dots, x_6 represent the complemented versions of the variables. We define the elements of S and the subsets C below. The translation to ILP form is now straightforward.

$$\begin{aligned} S_1 &= (x_4, x_2), S_2 = (x_2, x_3), S_3 = (x_1, x_6) \\ C_1 &= \{S_3\}, C_2 = \{S_1, S_2\}, C_3 = \{S_2\}, C_4 = \{S_1\}, C_5 = \{\emptyset\}, \\ &C_6 = \{S_3\} \end{aligned}$$

4. GENERIC ILP-BASED ENGINEERING CHANGE

In the modern design processes, there is often a need to handle small specification alterations and therefore to redesign or update the hardware and/or software implementations (EC). Until now, the EC methodologies and algorithms have been restricted to a particular task. In addition, EC

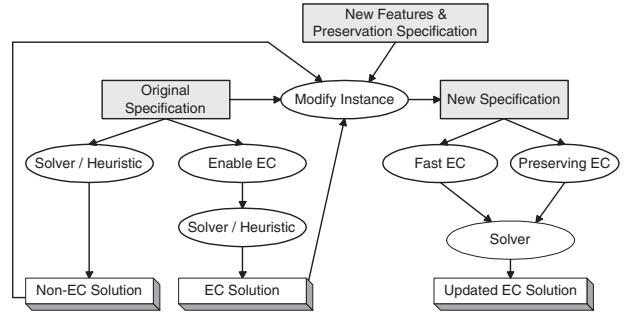


Figure 1: Generic ILP-Based EC Flow.

has been ad-hoc in nature is the sense that a few, if any, guarantees about its optimality could be provided. Our goal is to provide the first systematic, provably optimal, generic EC approach that can be easily applied to a variety of design and compilation tasks. The key enabling step of this effort is the formulation of EC requirements as ILP constraints. An important observation is that even when an optimization/synthesis problem is defined in ILP form, one can use not just an ILP solver, but also an arbitrary algorithm, such as simulated annealing or a heuristic, to solve it.

There are two ways how we can address EC requirements. One is to design for EC, where we embed flexibility into the solution to make it amenable for future changes or improvements. The other aspect is to apply EC after we have a valid and highly optimized solution. The goal here is to minimally alter the initial solution in order to obtain a solution to the altered specification. This task can be performed in at least two different setups: one where the goal is to preserve as much as possible from the original design (so the results of the consequent design steps are preserved maximally) and the other, where the speed of redesign is of prime importance.

Both types of EC require thoughtful considerations. For the first task, one should efficiently predict the parts of the design that would undergo alterations and updates and find mechanisms to make them flexible. For the second, the goal is to figure out how to specify a small and easy to solve instance of ILP (in the case of fast EC), or how to specify that the new solution to the new specification should preserve as much of the original as possible, or conserve the requested parts (in the case of preserving EC).

We now informally introduce the new EC approach. We conduct enabling EC in two different generic ways. The first one is to specify additional constraints that guarantee the solution will have flexibility as requested by the user. The second approach is that we add a new component, in terms of constraints, to the objective function. Now the objective function has two weighted components, the original part for the quality of solution and the new part for flexibility.

For fast EC, we follow the following procedure. We first isolate constraints that are impacted by the change. Using these constraints and new constraints, we specify a new instance of ILP. Finally, we combine the preserved parts of the initial solution and a partial new solution to create a complete new solution. For preserving EC, we define an ILP where either constraints or the objective function have components that guarantee preservation of either specified components of the previous solution, or as much as possible of the previous solution.

We conclude this section by presenting the generic ILP-based EC flow. The flow for a generic problem with EC can be seen in Figure 1. We begin with the original specification of the problem. We have two options. We can solve the instance with a standard ILP solver or the heuristic iter-

ative improvement-based ILP solver presented in [6], or we can apply enabling EC to the original problem specification. The result of the solver/heuristic on the original instance is denoted as the non-EC solution. Another option is to use an EC enabling procedure to produce the EC solution. Either the non-EC solution or the EC solution can be used in conjunction with the original problem specification and the new features and preservation specification to generate a new problem specification.

The new problem specification is a modified instance of the original problem specification that includes the original solution (EC or non-EC) and, if desired, the preservation specification. The EC or non-EC solutions are the starting solution for solving the new problem specification. We can resolve the new problem in two ways: fast EC or preserving EC. With fast EC, the new problem specification is simplified in such a way that the new ILP is much easier to solve. For preserving EC, we follow the preservation specification. Next, we find the solutions using a standard ILP solver. In this case, we do not use the heuristic ILP solver because the ILP solver will provide an optimal solution in a reasonable amount of time (the new specification is non-trivially smaller than the original instance).

5. ENABLING EC

In this section, we describe how to enable EC for the SAT problem by reformulating the ILP in such a way that we can adapt to changes in the problem.

For example, a variation of the SAT problem would be to remove new variables or clauses. When we remove variables from the problem, we modify the number of subsets, x , and therefore the number of columns, or subsets, in matrix A is changed. On the other hand, if we add clauses, the number of rows, or number of elements in S , in matrix A changes. Note that, in a sense adding clauses is a more general case than removing variables.

If we remove clauses or add variables to the problem we will not need to resolve the problem, because we are loosening the constraints on the problem. If we remove a variable or add clauses, we constrain the problem even more and the original solution is not necessarily adequate. By enabling EC we allow for the removal of variables or the addition of clauses such that their effects can be localized, and a minimum number of clauses and variables that are in direct relationship to them are changed.

One way to enable EC is to make sure that at least two variables per clause are satisfied. We call these clauses k -Satisfied, where k is the number of literals in the clause that evaluate to true. This can be very expensive or impossible in the general case. Therefore, we make a modification to the enabling condition. We say that we want to maximize the number of clauses that are at least 2-satisfiable. For all clauses that are 1-satisfiable there must be other literals in the clause that can switch their assignments to make the clause satisfied.

The ILP formulation for enabling EC is best introduced and explained using a small example. Consider again the SAT formula, F introduced above, along with its satisfying truth assignment. We label the clauses, c_1 , c_2 , and c_3 respectively. The ILP formulation is the same as presented in Section 3, except that we impose additional constraints in the following way.

For each clause we form a number of constraints. The first constraint is that each clause must be at least 2-Satisfiable (denoted by the first summation), or have at least one variable that can flip its assignment in such a way that it does not make the problem unsatisfiable (the second summation). Note that Z_i variables must evaluate to false in the considered clause in order to be included in the constraint.

$$\sum_i (x_i) + \sum_i (Z_i) \geq 2 \quad (7)$$

Recall that the constraint from (5) guarantees that at least one variable in each clause must be satisfied. A set of constraints is created for each inverse occurrence of each literal of the instance.

In our example, we begin with v_1 in c_1 . We say that it is permitted for variable v_1 to be eliminated from c_1 if either v_2 or v_3 can switch its assignment to satisfy the clause. We write equations for each occurrence of the complement of v_1 ; in our example it is v'_1 or x_5 in ILP form. Note that v'_1 appears only in c_3 .

We define two auxiliary variables, Q_i and Z_i . Z_i represents the variable that receives support from all variables in clause c_i . Q_i is used to ensure that all variables, Z_i , have non-negative values. Z_{ij} represents the support for variable x_i in clause c_j . Finally, we introduce variable Z_{ijk} which indicates whether variable x_i receives support from clause c_j through variable x_k when x_k flips its value.

The first two constraints specify that either v'_1 must satisfy the clause or one of the other literals in the clause must. We enforce this by creating a constraint that specifies that the complements of v_2 and v_3 must be greater than Z_{ijk} in the case that test variables are needed to satisfy the first two constraints. The fifth equation states that either one of the variables or none of them must be able to flip their value. If it is none of the variables, then Q_1 must be 1. Q_1 then is used in the next equation to specify that the value of variable Z_{53} must be zero. If this is the case, then Z_5 must be zero, and none of the variables can flip their value. From (7) we can see then at least two variables in c_1 must evaluate to true in order to satisfy the constraint. If either v'_2 or v_4 or both can flip their values then Z_{536} or/and Z_{534} will be 1. As a result then Z_{53} will be greater than zero, that results in equation (7) being satisfied.

$$\begin{aligned} x_5 + Z_{536} &\leq 1 \\ x_5 + Z_{534} &\leq 1 \\ x_6 &\geq Z_{536} \\ x_4 &\geq Z_{534} \\ Z_{536} + Z_{534} + Q &\geq 1 \\ Z_{536} + Z_{534} + Q - 1 &\geq Z_{53} \\ Z_5 &\leq Z_{53} \end{aligned}$$

By adding these constraints, we have now enforced that each clause must be either 2-Satisfied or has flexibility in terms of an alternate variable that can flip its assignment to support the unsatisfied clause, in case of an EC.

6. FAST SOLVING EC

While doing fast EC, there are two cases to consider for changing the constraints to the SAT problem. The first case is when variables are added or clauses are deleted. The other is when we add clauses or delete variables. The first case is trivial to handle in the sense that if a variable is added, it can automatically be assigned a DC value. The problem was originally satisfied and therefore the addition of a variable has no effect on the solution. The same is for the deletion of clauses. When clauses are deleted, the idea is to increase the enabling of the problem such that the next EC can be easily and properly handled.

We can increase the EC flexibility of the problem in two ways. First, we try and recover as many DC variables from the initial solution as possible. The second way is to reconstruct the solution in such a way that more clauses are of 2-satisfiability or higher.

If we add more clauses or remove variables, modifications must be made in order to quickly resolve the problem. We have developed the following approach, presented in Figure

2, that performs the minimization of the problem, in terms of the number of clauses and variables, in order to fast solve the instance.

<p>Input: F' a modified SAT Formula. p the original satisfying truth assignment to F.</p> <p>Output: A simplified SAT instance with the minimum number of clauses and variables.</p>
<p>Algorithm: <i>Check the Original Solution</i>(F', p); If p satisfies F' then quit; <i>Mark All Unsatisfied Clauses from</i> F'; Add all variables which appear in the marked clauses to list, V; <i>While</i> (V increases in size) { <i>Check All Clauses that have variables from</i> V If the clause is not satisfied by a variable not in V mark clause; add any new variables to V; } <i>Create new ILP, F'', with V and marked Clauses;</i> <i>Solve F'';</i> <i>Combine p and new solution p';</i></p>

Figure 2: Pseudo code for simplifying a SAT instance in order to be fast solved.

The intuition is the following. We solve the instance with all clauses that are not satisfied and the clauses that they could affect. If a clause is solved by other variables that need not be modified, we do not include them into consideration in the new SAT formula F' . If EC was enabled on the instance, then many clauses will not be included due to the fact that many of them will be 2-Satisfied or more. In this case, a minimal number of variables and clauses will have to be resolved.

7. PRESERVING EC

This section presents the ILP-based SAT formulation for preserving EC. We can preserve in two different ways. The first is to preserve the maximum amount possible, and the second is to preserve user specified parts of the solutions. This approach focuses on the quality of a solution rather than the overhead.

We only need to consider the case when variables are removed or clauses are added. In this case, we are removing constraints from the instance and therefore the problem is a simplified version of the original problem that the initial solution will satisfy.

When variables are removed or clauses are added we need to redefine the ILP such that the new solution, n , preserves as much as possible of the original solution, p . The key is that we formulate the ILP in such away that we are maximizing the number of assignments that stay the same as well as finding a satisfying truth assignment for the modified instance. We do this by adding additional constraints to the ILP or by modifying the objective function to give benefit to variable assignments which are preserved. The ILP for preserving EC is defined as follows.

$$\begin{aligned}
 x_i &= \begin{cases} 1, & \text{if subset } C_i \text{ is selected} \\ 0, & \text{otherwise.} \end{cases} \\
 N_i &= \begin{cases} 1, & \text{if subset } C_i \text{ is selected} \\ 0, & \text{otherwise.} \end{cases} \\
 A_{ij} &= \begin{cases} 1, & \text{if element } S_j \text{ is covered by subset } C_i \\ 0, & \text{otherwise.} \end{cases}
 \end{aligned}$$

The objective function is now defined as follows, where c is an identity vector. In this case, we are maximizing the

number of variables which have the same assignment as the original assignment, p . We keep the same constraints (5) and (6), where b is an identity vector. We add the following constraint that evaluates whether or not the new variable assignment for N_i is the same as the assignment as p_i .

$$\begin{aligned}
 Y &= \text{MAX}(cZ) \\
 Z_i &= p_i x_i + p_{n+i} x_{n+i}
 \end{aligned}$$

8. EXPERIMENTAL RESULTS

We use standard DIMACS SAT benchmarks to test our EC methodology. We use CPLEX as our main ILP solver. Both CPLEX and the heuristic ILP solver were ran on a 1 GHz Pentium III computer.

The results for applying EC to DIMACS SAT instances are shown in Table 1. The first three columns indicate the name of the instance, the number of variables and the number of clauses in the instance, respectively. The next column presents the runtime for the original instance. The fifth column represents the runtime values when specified constraints are imposed. The last column shows the runtime when the objective function is augmented with EC requirements. Both of these columns represent the normalized runtime against the original runtime. The average and median for the specified constraints are 0.69 and 2.62 respectively and for objective function EC are 0.75 and 2.34 respectively. As we can see, the overhead is not significant. In all the tests, we used $k = 2$, that means that each clause is directly either 2-Satisfied or is satisfied by the support of other variables in the clause being able to flip their assignment without jeopardizing the satisfiability of other clauses. The second set of results, the five examples shown on the bottom of the table, are solved heuristically using the heuristic ILP solver [6].

For fast solving, we ran ten trials. For each of the trials, we eliminated three variables and added ten clauses. The results are shown in Table 2.

The first three columns represent the DIMACS instance information. The third column indicates the original runtime for solving the instance. The following two columns list the average number of variables and clauses in the fast EC instance. The last column represents the runtime of the fast EC instance. On average, for the smaller examples, we needed 23.25 variables and 99.6 clauses, and on average reduced the runtime to 0.0068 of the original runtime. For the larger examples at the bottom of the table, we initially solved them using our heuristic ILP solver. Once an initial solution was generated, we then used an off-the-shelf solver to obtain a new optimal solution to the original instance.

To evaluate our preserving EC, we randomly added and deleted five variables and randomly added and deleted five clauses, making sure that we did not make the instance non-satisfiable. We compared the percentage of preserved variable assignment for two cases: when new instance were just again evaluated with no consideration for preserving the initial assignment as when preserving EC approach is used. We present the results in Table 3. We begin the table with the DIMACS instance information, name, number of variables and number of clauses in the instance. The fourth column indicates the percentage of the original solution after complete recalculation with no EC goals. The percentage of the original solution preserved when evaluating the instance with preserving EC is shown in the last column. The last two rows present the average and median values. It is clear that preserving EC preserve significantly higher percentage of the initial assignment. As was done for fast solving, the larger instances presented at the bottom of the table were originally ran using the heuristic ILP solver to generate the initial solution. An off-the-shelf solver was used to optimally solve the instance with preserving EC.

In addition to validating the new ILP-based engineering

change approach on SAT benchmarks, we conducted comprehensive experimentation on the graph coloring problem. The results can be found in [6].

9. CONCLUSION

We introduced a generic ILP-based EC methodology. We demonstrated the three components, enabling EC, fast EC, and preserving EC on a CAD related problems, SAT. The EC techniques are applicable to many other CAD and optimization problems. We demonstrated the effectiveness of the methodology on standard DIMACS benchmark instances.

10. ACKNOWLEDGEMENTS

This research was supported by the NSF CARRIER award 9734166 at UCLA. Farinaz Koushanfar is supported by a NSF graduate research fellowship.

11. REFERENCES

- [1] M. Garey and D. Johnson. *Computers and Intractability*. W.H. Freeman, 1979.
- [2] C.H. Gebotys and M.I. Elmasry. Global optimization approach for architectural synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(9):1266–1278, September 1993.
- [3] J. Gu, P.W. Purdom, J. Franco, and B.W. Wah. Algorithms for the satisfiability (sat) problem: a survey. In *Satisfiability Problem: Theory and Applications. DIMACS Workshop*, pages 19–51, 1997.
- [4] S.P. Khatri, A. Narayan, S.C. Krishnan, K.L. McMillan, et al. Engineering change in a non-deterministic fsm setting. In *Proceedings of IEEE/ACM Design Automation Conference*, pages 451–456, June 1996.
- [5] D. Kirovski and M. Potkonjak. Engineering change: methodology and applications to behavioral and system synthesis. In *Proceedings of IEEE/ACM Design Automation Conference*, pages 604–609, June 1999.
- [6] F. Koushanfar, J.L. Wong, J. Feng, and M. Potkonjak. Ilp-based engineering change. In *Technical Report, CS Department, UCLA*, pages 1–11, April 2001.
- [7] C. Lin, K. Chen, and M. Marek-Sadowska. Logic synthesis for engineering change. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(3):282–292, March 1999.
- [8] M. Lin, H. Perng, C. Hwang, and Y. Lin. Channel density reduction by routing over the cells. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 120–125, June 1991.
- [9] J.P. Marques-Silva and K.A. Sakallah. Boolean satisfiability in electronic design automation. In *Proceedings of IEEE/ACM Design Automation Conference*, pages 675–680, July 2000.
- [10] C.H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity*. New Jersey: Prentice Hall, 1982.
- [11] H.M. Salkin. *Integer programming*. Massachusetts: Addison-Wesley, 1975.
- [12] A. Schrijver. *Theory of linear and integer programming*. New York: Wiley, 1986.
- [13] T. Shinsha, T. Kubo, Y. Sakataya, J. Koshishita, and K. Ishihara. Incremental logic synthesis through gate logic structure identification. In *Proceedings of IEEE/ACM Design Automation Conference*, pages 391–397, June 1986.
- [14] L.A. Wolsey. *Integer Programming*. New York: Wiley, 1998.

Instance	# Vars	# Clauses	Orig. Runtime	EC (SC) N.R.	EC (OF) N.R.
par8-1-c	64	254	21.14	0.90	2.12
ii8a1	66	186	0.14	0.82	1.37
par8-3-c	75	298	57.9	0.81	1.78
jnh201	100	800	527.83	0.77	2.28
jnh1	100	850	1476.59	0.73	2.40
ii8a2	180	800	3023.94	0.67	3.01
ii8b2	576	4088	20089.8	0.45	4.42
f600	600	2550	18989.8	0.37	3.56
average	-	-	5523.393	0.69	2.62
median	-	-	1002.21	0.75	2.34
par32-5-c	1339	5350	5.2	0.77	1.93
ii16a1	1650	19368	12.6	0.82	2.41
par32-5	3176	10325	22.4	0.94	3.87
g250.15	3750	233965	74.8	0.98	4.37
g250.29	7250	454622	96.9	0.92	3.98
average	-	-	42.38	0.88	3.31
median	-	-	32.39	0.90	3.59

Table 1: Experimental Results for Enabling EC on SAT.

Instance	# Vars	# Clauses	Orig. Runtime	Ave. # Vars/Clauses	New Runtime
par8-1-c	64	254	201.14	11.2/40.8	0.036
ii8a1	66	186	0.14	10.8/27.3	0.005
par8-3-c	75	298	57.9	16.0/38.5	0.007
jnh201	100	800	527.83	21.0/98.9	0.002
jnh1	100	850	1476.59	17.7/67.3	0.001
ii8a2	180	800	3023.94	25.7/165.7	0.002
ii8b2	576	4088	20089.8	56.4/191.4	0.001
f600	600	2550	18989.8	27.2/167.0	0.001
average	-	-	5523.393	23.25/99.61	0.007
median	-	-	1002.21	19.35/83.1	0.002
par32-5-c	1339	5350	5.2	52.6/387.2	261.2
ii16a1	1650	19368	12.6	68.1/401.6	76.2
par32-5	3176	10325	22.4	70.9/476.2	102.9
g250.15	3750	233965	74.8	74.3/639.1	202.7
g250.29	7250	454622	96.9	102.5/876.4	952.1
average	-	-	42.38	73.68/1416.1	319.02
median	-	-	32.39	72.29/757.75	202.7

Table 2: Experimental Results for fast EC on SAT.

Instance	# Vars	# Clauses	% Solution Original	% Solution with EC
par8-1-c	64	254	72.2	98.2
ii8a1	66	186	71.6	99.3
par8-3-c	75	298	68.2	94.7
jnh201	100	800	63.6	93.7
jnh1	100	850	72.8	98.2
ii8a2	180	800	82.7	99.4
ii8b2	576	4088	83.0	99.7
f600	600	2550	76.6	98.6
par32-5-c	1339	5350	62.4	92.8
ii16a1	1650	19368	73.5	99.3
par32-5	3176	10325	71.6	94.1
g250.15	3750	233965	68.7	97.8
g250.29	7250	454622	84.1	94.7
average	-	-	73.19	96.96
median	-	-	72.75	98

Table 3: Experimental Results for preserving EC on SAT.