Exploiting Constraint Solving History to Construct Interaction Test Suites

Myra B. Cohen, Matthew B. Dwyer, Jiangfan Shi
Department of Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, Nebraska
{myra,dwyer,jfshi}@cse.unl.edu

Abstract

Researchers have explored the application of combinatorial interaction testing (CIT) methods to construct samples to drive systematic testing of software system configurations. Applying CIT to highly-configurable software systems is complicated by the fact that, in many such systems, there are constraints between specific configuration parameters that render certain combinations invalid. In recent work, automated constraint solving methods have been combined with search-based CIT methods to address this problem with promising results.

In this paper, we observe that the pattern of computation in greedy CIT algorithms leads to sequences of constraint solving problems that are closely related to one another. We propose two techniques for exploiting the history of constraint solving: (1) using incremental algorithms that are present within available constraint solvers and (2) mining constraint solver data structures to extract information that can be used to reduce the CIT search space. We evaluate the cost-effectiveness of these reductions on four real-world highly-configurable software systems and on a population of synthetic examples that share the characteristics of those four systems. In combination our techniques reduce the cost of CIT in the presence of constraints to that of traditional unconstrained CIT methods without sacrificing the quality of solutions.

1. Introduction

Combinatorial interaction testing (CIT) has proven to be an effective technique for systematically sampling a program's input space to achieve a high-degree of coverage and fault detection [3, 15, 24]. When using CIT one models a system as a collection of *factors* each with a finite set of possible *values*. The technique produces a set of factor-value bindings that, typically, cover all possible pairs of factor-values; higher-order coverage can also be achieved. CIT is attractive because by covering all pairs or all n-tuples it is possible to detect errors that arise only when specific combinations of factor values are enabled in a system.

Factors are often used to model program inputs and their values to model specific input values, or equivalence classes of values [3, 9]. In recent work, researchers have adapted CIT techniques to reason about highly-configurable software systems, i.e., one's that have a large number of options for enabling and disabling different system capabilities [15, 24]. CIT models for such systems would encode an option as a factor and the possible choices for the option as values. For example, Firefox allows users who enable cookies to keep those cookies until they expire, until the browser is closed, or to not store cookies and ask the user to confirm acceptance of every cookie; this would be modeled as a single factor with three possible values. Validating such systems presents some significant challenges over and above traditional software systems. The problem of testing a single software system has been replaced with the much harder problem of testing the *set* of software systems that can be produced by all of the different possible bindings of options. A single test case may run without failing under one configuration, however the same test case may fail under a different one [15, 24]. One cause for this is the unintended *interaction* of multiple optional capabilities.

The optional capabilities of highly-configurable systems are rarely completely independent. For example, the GCC [11] man pages describe numerous options whose settings are dependent on the setting of other options, e.g., "-finline-functions-called-once ... Enabled if -funit-at-a-time is enabled." and "-fsched2-use-superblocks ... This only makes sense when scheduling after register allocation, i.e. with -fschedule-insns2". Failing to take such option constraints into account is problematic for multiple reasons. Ignoring constraints may lead to the generation of test configurations that are illegal and this can lead to inaccurate test planning and wasted effort. Even a small number of constraints can give rise to enormous numbers of illegal configurations. In [6] we report on a configuration model for the GCC 4.1 optimizer that consists of 199 factors and 40 constraints; those constraints render about 25%, or 1.2×10^{61}, of the possible optimizer configurations illegal. Clearly, it is infeasible to enumerate the illegal configurations. An additional challenge arises because multiple *explicit* constraints may interact to *imply* additional constraints; in our GCC model two such implicit constraints were present. Even if a system were small enough to allow enumeration of configurations, accounting for the interactions among constraints requires non-trivial reasoning.

To address CIT in the presence of constraints, we extended, in [6], the theoretical basis of CIT to include general constraints encoded as propositional formulae defined over factor-value bindings and presented a general approach to integrating propositional boolean satisfiability (SAT) solvers with greedy and search-based methods for CIT. This method demonstrated that it is feasible to treat the types of constraints that arise in real-world configurable systems in existing CIT algorithms. Evaluation on a collection of realistic case studies and smaller examples demonstrated that it is essential to consider constraints, since unconstrained CIT produces invalid test configurations that lead to wasted effort in subsequent test planning and development. While feasible, however, the integration of SAT and CIT algorithms increased the already significant cost of CIT algorithms by more than 56% across three realistic case studies. This is cause for concern about the scalability of the technique.

In this paper, we seek to drive down the cost of solving CIT problems in the presence of constraints without sacrificing solution quality, i.e., the number of generated test configurations. To achieve this, we investigate approaches for a synergistic integration of greedy CIT algorithms and SAT algorithms. An effective integration is possible because greedy CIT algorithms incrementally construct a set of candidate factor-value bindings and this produces sequences of SAT solver calls on formulae that are closely related to one another. More specifically, every formula in the sequence is an extension of an earlier formula where a single additional conjunct is added. This *history* of SAT solver calls paves the way for several optimizations to reduce CIT cost in the presence of constraints. The findings we report in this paper make several contributions: (*i*) we investigate the cost-effectiveness of incremental SAT algorithms to exploit SAT history in CIT, (*ii*) we present a technique for mining internal SAT solver data structures to prune the CIT algorithm's search space, (*iii*) we extend our study of real highly-configurable software systems to include SPIN [14], GCC [11], Apache [1], and Bugzilla [20], and (*iv*) we report the results of an evaluation that demonstrates the cost-reduction and solution-quality-preservation of our techniques across a range of configuration models.

In the next section, we provide some background on interaction testing, modern SAT solvers, and an existing approach to combining the two. Section 3 provides a more detailed presentation of the structure of SAT history that we exploit, then describes how incremental SAT solving is applied to CIT and, finally, how we exploit boolean constraint propagation (BCP) [18] information calculated during SAT to reduce subsequent CIT processing. Section 4 provides an overview of the real systems we studied, outlines our methodology for synthesizing configuration models that share the characteristics of those systems, and then reports on the cost and solution quality of different combinations of CIT and SAT techniques. In Section 5 we conclude and discuss future work.

2. Background

To illustrate the challenges of testing highly-configurable software consider GCC 4.1 [11]. In our analysis of the manual pages for GCC [6], we found that it has 1462 different run-time options that control the phases included in the compilation process and the functionality of those phases. 199 of those options are related to the global optimization phase; 186 of those are boolean configuration parameters and the remaining 10 parameters have three different settings. In total, there are $2^{186} \times 3^{10}$, or 4.6×10^{61}, different configurations of the GCC optimizer. Thorough testing of the optimizer is critical, but it is clearly impossible to test all configurations.

2.1. Combinatorial Interaction Testing

One strategy that has been applied is to systematically sample instances of software configurations so that all t-way combinations of options appear [3]; we call this *CIT sampling*. Testing those configurations has the potential to expose errors that arise due to the t-way interaction of configuration-specific components.

CIT is widely regarded as a powerful sampling technique for functional input testing that may increase the ability to find certain types of faults efficiently [3] and that provides good code coverage [3, 9]. A major focus in the literature has been the development of efficient algorithms to find smaller t-way samples [4, 12, 13, 21, 23]. Recent work has seen those algorithms applied to user configurable systems [15, 24], and software product lines [5, 19].

2.1.1 CIT Samples

A t-way CIT sample is defined by a *covering array* [4]. As described above, the GCC optimizer has some configuration parameters that have two values and some with three values. We use a *mixed level covering array* to accommodate variation in the number of values.

```
mAETG(CAModel)
Require: $uncovered\text{-}t\text{-}set\text{-}count$: calculated by initialization
 1: $numCandidates = 50$
 2: $numTests = 0$
 3: $testCasePool = \emptyset$
 4: while $uncovered\text{-}t\text{-}set\text{-}count > 0$ do
 5:     for $count = 1$ to $numCandidates$ do
 6:         $testCase_{count}$=generateEmptyTestCase()
 7:         $l$=selectFirstFactorValue($unCovSet$)
 8:         $f$=selectFirstFactor($l$)
 9:         insertValueForFactor($l$,$f$,$testCase_{count}$)
10:         $p$=permuteRemainingFactors()
11:         for $f \in p$ do
12:             $l$=selectBestValue($f$)
13:             insertValueForFactor($l$,$f$,$testCase_{count}$)
14:         saveCandidate($testCasePool$,$testCase_{count}$)
15:     selectBestCandidate($testCasePool$)
16:     update($uncovered\text{-}t\text{-}set\text{-}count$)
17:     increment $numTests$
```

Algorithm 1: AETG Algorithm

Definition 2.1 *A mixed level covering array, $MCA(N; t, k, (v_1, v_2, ..., v_k))$, is an $N \times k$ array on v symbols, where $v = \sum_{i=1}^{k} v_i$, with the following properties: (1) Each column $1 \leq i \leq k$ contains only elements from a set S_i of size v_i. (2) The rows of each $N \times t$ sub-array cover all t-tuples of values from the t columns at least one time [4].*

The k columns of this array are called *factors*, where factor f_i has v_i *values*. A covering array model, *CAModel*, consists of k and v_i, $i \leq 1 \leq k$. We use a shorthand notation to describe mixed level covering arrays by combining entries with equally sized value domains v_i. For example, a 2-way CIT sample for the GCC optimizer would be written as an $MCA(N; 2, 2^{189}3^{10})$ and its CAModel would be $2^{189}3^{10}$.

2.1.2 Finding CIT Samples

Finding a covering array for a configurable system is an optimization problem where the goal is to find a minimal set of configurations satisfying the *coverage* criteria of all t-sets. Many algorithms and tools exist that construct covering arrays, but we focus in this paper on one-row-at-a-time greedy-algorithms in the style of the automatic efficient test case generator (AETG) [3]. Multiple variants of AETG have appeared in the literature, e.g., [7, 22], and we refer to these as *AETG-like*.

Algorithm 1 sketches the basic structure of this algorithm. Prior to execution an initialization step is used to calculate the number of t-sets for the given problem; covering all such sets drives continued execution of the algorithm. The algorithm constructs an array with `numTests` rows. A single row for the array is constructed in each iteration of the loop at line 4 until all t-sets have been covered. The algorithm constructs `numCandidates` different rows, line 5, and selects the best one to add to the array, lines 15-17. The choice of the size of candidate set is one of the differentiators of AETG-like algorithms. Our algorithm uses the value 50 for `numCandidates` to be consistent with the original description of AETG [3].

```
mAETG-SAT(CAModel)
Require: $uncovered\text{-}t\text{-}set\text{-}count$: calculated by initialization
 1: $numCandidates = 50$
 2: $numTests = 0$
 3: $testCasePool = \emptyset$
 4: while $uncovered\text{-}t\text{-}set\text{-}count > 0$ do
 5:     for $count = 1$ to $numCandidates$ do
 6:         $testCase_{count}$=generateEmptyTestCase()
 6a:        $sat$=false
 6b:        while !$sat$
 7:             $l$=selectFirstFactorValue($unCovSet$)
 8:             $f$=selectFirstFactor($l$)
 8a:            $sat$=$\neg$ factorInvolved($f$) $\vee$ checkSAT($testCase_{count}$)
 9:         insertValueForFactor($l$,$f$,$testCase_1$)
10:         $p$=permuteRemainingFactors()
11:         for$f \in p$ do
11a:            $sat$=false
11b:            $tries = 1, maxTries = v$
11c:            while !$sat$ and $tries \leq maxTries$
12:                 $l$=selectBestValue($f$)
12a:                $sat$=$\neg$ factorInvolved($f$) $\vee$ checkSAT($testCase_{count}$)
12b:                increment $tries$
13:             insertValueForFactor($l$,$f$,$testCase_{count}$)
14:         saveCandidate$TestCasePool$,$testCase_{count}$)
15:     selectBestCandidate($testCasePool$)
16:     update($uncovered\text{-}t\text{-}set\text{-}count$)
17:     increment $numTests$
```

Algorithm 2: AETG-SAT Algorithm

To build a single row, heuristics are applied to select the first factor and its value, lines 7-9. In AETG a factor-value pair is chosen that currently has the largest number of t-sets left to cover. The order in which the remaining factors are processed is shuffled, line 10, and then the best value for each factor is selected, line 12-13, where the best value produces the most previously uncovered t-sets. Other greedy algorithms [7, 22] use slightly different heuristics to select the factor ordering.

2.2. Constrained CIT (CCIT)

In [6], we identify the need to treat constraints in CIT and present an extension of the CIT model and algorithms to find CIT samples in the presence of constraints. We briefly review the key concepts here.

Constraints may disallow combinations of options, which we refer to as *forbidden* constraints, or require that when one option value is selected that another also be selected. In this paper, we convert all constraints into a set of forbidden constraints. We refer to these as *explicit* constraints, but it is known that combinations of explicit constraints can interact to give rise to *implicit* constraints [6].

Constraints are encoded as boolean formulae defined over propositional variables that encode factor-value pairs; a boolean option encodes a single variable, but an option with n values will have n variables.

Explicit forbidden constraints are naturally encoded as the negation of a conjunction of the propositional variables for factor-values. Sometimes other constraints require transformation into this form. For example, the GCC optimizer constraint "-finline-functions-called-once ... Enabled if -funit-at-a-time is enabled." expresses an implication between the enabling of "unit-at-a-time" and "inline-functions-called-once". Negating this implication results in a forbidden constraint between *inline-functions-called-once=false* and *unit-at-a-time=true*. For example, this forbidden constraint would be expressed as (*inline-functions-called-once* $\vee \neg$ *unit-at-a-time*). Let A be the set of all forbidden constraints expressed as disjunctions of propositional terms.

We require that each factor in the covering array has a value in each row and this is encoded by *at-least* constraints in our model [13]. For each factor, $f \in f_1 \ldots f_k$, an at least constraint is simply $al_f \equiv \bigvee_{v \in V_f} f = v$; where V_f are the possible values for factor f. While not strictly required, and therefore not included in our approach in [6], an additional set of *at-most* constraints [13] ensures that each factor has a single value. For each factor, $f \in f_1 \ldots f_k$, an at-most constraint is simply $am_f \equiv \bigwedge_{v,v' \in V_f \wedge v \neq v'} f \neq v \vee f \neq v'$. We refer to

$$(\bigwedge_{a \in A} a) \wedge (\bigwedge_{f \in f_1 \ldots f_k} al_f) \wedge (\bigwedge_{f \in f_1 \ldots f_k} am_f)$$

as the common base constraints, C, for a constrained CIT problem; as defined C is in conjunctive normal form.

Treatment of constraints requires a modified definition of a CIT sample.

Definition 2.2 *A* constrained mixed-level covering array*, $CMCA(N; t, k, (v_1, v_2, ..., v_k), C)$ is an $N \times k$ array on v symbols with constraints C, where $v = \sum_{i=1}^{k} v_i$, with the following properties: (1) Each column $1 \leq i \leq k$ contains only elements from a set S_i of size v_i. (2) The rows of each $N \times t$ sub-array cover all C-consistent t-tuples of values from the t columns at least one time.*

A tuple, is C-consistent if it can be extended to a row, i.e., a k-set, r, and $C \wedge r$ is satisfiable. Note that such a satisfiability test naturally accounts for implicit constraints so they need not be included in C.

Algorithm 2 illustrates the integration of C-consistency checks into our AETG-like algorithm where the *CAModel* has been extended to include C. The algorithm is modified in three areas. (1) We piggy back onto the initialization step (not shown) a calculation of the set of factors that are involved in some constraint - binding values for uninvolved factors does not require a consistency check. (2) If a consistency check fails, we must undo a factor value binding and try another; lines 6a-6b and 11a-11c and 12b realize this backtracking. (3) Consistency checks, lines 8a and 12a, are introduced to determine if the extension of the row is consistent with the constraints. If we reach *maxTries* without reaching a satisfiable solution, the test candidate is removed from the potential set of solutions (not shown). In principle, any modern SAT solver could be used to discharge consistency checks - in previous work we used zChaff [17] since it is regarded as an efficient solver. We note that this integration, while effective on previous case studies [6], amounts to an approach where AETG *generates* a candidate row using its heuristics and SAT *tests* it for consistency.

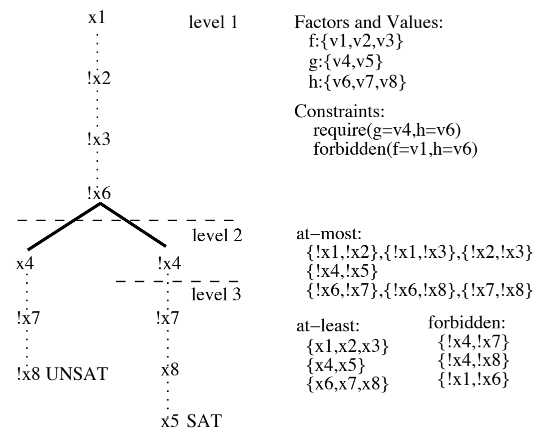

Figure 1. Example SAT Search

2.3. Boolean Satisfiability Solving

In Section 3, we will present a synergistic integration of AETG and SAT solving algorithms. To understand this integration a basic understanding of modern SAT solving algorithms is required.

SAT solvers work on formulae encoded in conjunctive normal form (CNF). A CNF formula is a set of clauses each of which must be true for the formula to be true. Clauses in turn are disjunctions of a set of propositional variables or their negation; we write such clauses using set notation, e.g., $\{x1, !x2, x3\}$ denotes the clause $x1 \vee \neg x2 \vee x3$.

State-of-the-art SAT solvers are based on the classic DLL backtracking search [8] that explores a tree of truth-assignments for propositional variables. There is a rich literature on extensions to this algorithm to scale it to the point where satisfiability can be checked on formulae with many tens of thousands of variables. We discuss two techniques that have been widely adopted in the SAT community: boolean constraint propagation (BCP) and conflict-clause learning [18].

Figure 1 illustrates a simple CCIT satisfiability check for a system with one binary, g, and two ternary, f and h, factors. Two constraints are also specified: a require constraint, which is converted into two forbidden constraints, and a single forbidden constraint. As explained above, the satisfiability problem is expressed in terms of distinct propositional variables for each possible factor-value assignment, e.g., $f = v3$ is modeled with $x3$. The figure shows two branches of a search for a satisfying assignment given the input clause $x1$, i.e., $f = v1$. The search exploits the CNF clauses shown on the lower right of the figure.

SAT solvers divide the process into two alternating phases: *search* and *propagation*. A search phase (denoted by solid edges) involves the selection of a propositional variable and a truth assignment for it; both may be informed by heuristics. A propagation phase (denoted by consecutive dotted edges) involves using the current partial truth assignment, defined by the path in the search tree, and the set of CNF clauses to infer the values of propositional variables.

The BCP process attempts to produce *unit clauses* in order to force a truth assignment to a variable. A unit clause is one that has a single unbound variable in it. Since all clauses must be true for the CNF formula to hold, the polarity of a variable in a unit clauses implies its truth assignment. In the Figure, the first step is an implicit search step reflecting the assignment of $x1$ to true. The next three steps arise from BCP. For example, in order for the clause $\{!x1, !x2\}$ to be true when $x1$ is true, it must be the case that $!x2$. After $!x6$ is assigned the algorithm performs a search step where it fixes $x4$; each search step increases the *level* of the search. A second BCP phase follows where $!x7$ and $!x8$ are assigned. At this point, the formula can be determined to be unsatisfiable since the clause $\{x6, x7, x8\}$ is false and the search backtracks. Satisfiability requires a single truth assignment to be found and the right branch illustrates such an assignment; at this point the search stops with $x1$, $x5$, and $x8$ true.

In general, when the search backtracks some subset of the truth assignment is responsible for the conflict that leads to the UNSAT result. In the case of our simple example, the combination of $x4$ and $!x6$ will always lead to a conflict with the clause $\{x6, x7, x8\}$. Conflict-clause learning techniques perform a dependence analysis of the sequence of truth assignments and the clauses that influenced those assignments in order to infer a minimal implicate for the conflict, i.e., the weakest clause that implies that the conflict is guaranteed to arise. The negation of the conjunction of the conflicting terms can be recorded by the solver and used to prevent subsequent searches from ever exploring truth assignments that falsify the clause. In our example, the clause $\{!x4, x6\}$ assures that the search will never fail for the same reason as it did along the left branch.

3. Exploiting SAT History

SAT solvers are designed to check satisfiability of formulae independently. Solvers, in general, do not exploit information from past searches in order to speed the search for a satisfiable assignment because they have no way of knowing the relationship between formulae submitted to the solver at different times.

Execution of Algorithm 2 produces a series of SAT calls on formula that are closely related to one another. The loop beginning at line 11 processes a row one factor at a time and in each iteration it assigns a value for the current factor. As discussed in Section 2.2, all formulae for a CCIT problem have a common set of *base constraints*, C, that are conjoined with the partial configuration being built for the row. The formula constructed on the k^{th} iteration of the loop is:

$$C \wedge f_1 = v_1 \wedge f_2 = v_2 \wedge \ldots \wedge f_k = v_k$$

using individual distinct atomic propositions to encode each $f_i = v_i$; we give distinct names to all v_i and use a propositional variable x_i to denote whether that value is bound to its associated factor. If the `checkSAT` call on line 12a succeeds then on the $(k+1)^{st}$ iteration of the loop, the formula

$$C \wedge f_1 = v_1 \wedge f_2 = v_2 \wedge \ldots \wedge f_k = v_k \wedge f_{k+1} = v_{k+1}$$

will be checked for satisfiability. The description of SAT solving in Section 2.3 makes it clear that checking this formula for satisfiability *could* restart the search at the point where it assigned x_k to true, then assign x_{k+1} to true and continue the search for a satisfiable assignment.

These observations led us to investigate three techniques for integrating AETG and SAT algorithms that we describe in the remainder of this section.

3.1. Adding constraints to enhance BCP

The SAT literature discusses the benefits and risks of adding additional clauses to a SAT problem [17, 18]. In general, more clauses gives BCP the opportunity to make more assignments to propositional variables in the propagation phase. On the downside BCP has to traverse more clauses, thereby potentially slowing the propagation phase. If clauses eliminate search phases, then they are likely to yield a significant benefit since avoiding search of a combinatorially sized sub-space of variable assignments more than compensates for a slight increase in BCP cost.

Figure 1 illustrates the potential benefit of at-most clauses in the first propagation phase by fixing the values of $x2$ and $x3$ to be false. If a forbidden constraint of the form $\{x3, !x7\}$ had been included, then propagation would have continued by inferring $!x7$, which in turn would have forced $x4$ to be false and the entire search for a satisfiable assignment would have finished in a single propagation phase.

In previous work [6], we followed the conventional wisdom of minimizing clauses in encoding our constraints. We observed that the nature of the AETG algorithm would allow us to eliminate at-most constraints without compromising the correctness of the solution. What we didn't understand was the impact this would have on reducing the effectiveness of BCP. While not an explicit topic of our evaluation, we have observed small performance improvements with the addition of at-most constraints.

3.2. Incremental SAT

Several modern SAT solvers offer support for adding and retracting clauses from a SAT problem rather than submitting a formula as a monolithic structure. When combined with conflict-clause learning, this allows for a type of incremental SAT solving where conflict-clauses learned in one SAT search can be used to prune a subsequent SAT search. The key issue here is the dependence of the conflict-clauses on any retracted clauses. A SAT solver like zChaff will perform a clause dependence calculation and remove both the retracted and any dependent learned clauses; this can be an expensive process. A less costly approach is supported by the MiniSAT [10] solver. MiniSAT allows a set of clauses to be passed as *assumptions* and its conflict clause learning algorithm only stores clauses that are not dependent on the assumption clauses; thus there is no cost for retracting clauses from assumptions.

We adapted Algorithm 2 to use incremental support in MiniSAT by adding clauses for the base constraints and incorporating the clauses encoding the partial configuration as an assumption. In this way, learned conflict clauses related to base constraints are accumulated across *all* SAT calls in a CCIT problem. As shown in Section 4, incremental SAT solving leads to non-trivial reductions in CCIT sample generation times relative to our previous implementation.

3.3. Mining SAT Assignment Information

To judge a partial configuration consistent with the CCIT constraints a SAT solver constructs a truth assignment for propositional variables. That assignment includes the values that are specified in the partial configuration, but it may also include additional values. Figure 1 showed how BCP propagation can assign values. In fact, the path from the root down to the node marked SAT encodes a definitive truth assignment for all eight propositional variables. Recall that the example checked the satisfiability of a partial configuration that fixed a single factor's value, i.e., $f_1 = v1$ which is encoded as $x1$, yet the SAT solver effectively calculated a total configuration. We exploit this information to *feed back* definitive factor-value bindings to AETG so that it can either (a) skip assigning a value to a factor later in a row when a factor-value variable is determined to be true by SAT or (b) reduce the set of possible values that could be assigned to a factor later in the row by eliminating values determined to be false by SAT.

Conceptually, this process is simple. When a formula is determined to be satisfiable, we record the truth assignment and return it to AETG along with the SAT verdict. An additional subtlety arises because, unlike in Figure 1, the truth assignment to determine satisfiable need not be total. Imagine a propositional variable that is not present in any clause – its value does not influence the satisfiability of the formula so there is no reason to infer that it is either *true* or *false*. Thus, our solution mines SAT data structures to return an indication of whether a variable is *true*, *false*, or *undefined* as a result of the preceding SAT call.

Algorithm 3 illustrates the extensions to Algorithm 2, shown in bold, that realize the feedback from SAT to AETG. We define two methods that mine SAT solver data structures. (1) `mineMayAssignments` returns, for each factor, the possible value assignments that are consistent with the current partial configuration as determined by SAT. It does this eliminating the possibility of assignments that SAT has determined must be *false*. Over a sequence of SAT calls the set of possible values for a factor decreases until a value is selected. (2) `mineMustAssignments` returns the set of factor-value pairs that must be present in the partial configuration as determined by SAT. It does this by extracting SAT assignments that must be *true*. Note that *undefined* values provide no information to AETG.

May assignment information is calculated, at line 11d, and used, at line 12, to prune the set of possible values from which AETG will select its best value. This greatly reduces the chance of selecting a value that will lead to an inconsistent partial configuration. Note that it is still possible for the algorithm to produce unsatisfiable partial configurations. This is because the mined assignment information is based only on the current partial configuration. For example, a variable assignment may be currently *undefined* while in a subsequent SAT call for an extension of the configuration that variable may be forced to be *false*.

Must assignment information is calculated, at line 13a, and used, in lines 13b-13d, to make additional factor-value assignments that ensure satisfiability of the current partial configuration. When a factor is assigned a value at line 13c, we delete that factor from p, at line 13d, so it is not explicitly targeted for a value assignment by the loop at line 11.

Note that the SAT solver produces truth assignments without regard for AETG's objective of covering all t-sets. Consequently, it is possible for must assignment information mined from SAT to force factor-value assignments that make it difficult for AETG to achieve full t-set coverage. We have not observed this in practice, in fact, as shown in Section 4 feeding back assignment information from SAT to AETG appears to produce similar size CMCA arrays as alternative techniques.

4. Evaluation

In this section we present a set of experiments designed to evaluate the effect of using the history based SAT solving technique when incorporated with a greedy AETG-like algorithm for finding CCIT samples. We begin with a description of the constrained covering array models, which

```
mAETG-History(CAModel)
Require: $uncovered\text{-}t\text{-}set\text{-}count$: calculated by initialization
 1:  $numCandidates = 50$
 2:  $numTests = 0$
 3:  $testCasePool = \emptyset$
 4:  while $uncovered\text{-}t\text{-}set\text{-}count > 0$ do
 5:    for $count = 1$ to $numCandidates$ do
 6:      $testCase_{count}$=generateEmptyTestCase()
 6a:     $sat$=false
 6b:     while !$sat$
 7:        $l$=selectFirstFactorValue($unCovSet$)
 8:        $f$=selectFirstFactor($l$)
 8a:       $sat$=$\neg$ factorInvolved($f$) $\vee$ checkSAT($testCase_{count}$)
 9:      insertValueForFactor($l$,$f$,$testCase_{count}$)
10:      $p$=permuteRemainingFactors()
11:      for $f \in p$ do
11a:       $sat$=false
11b:       $tries = 1, maxTries = v$
11c:       while !$sat$ and $tries \leq maxTries$
11d:         $maySet$=mineMayAssignments()
12:          $l$=selectBestValueFromMaySet($f$,$maySet$)
12a:         $sat$=$\neg$ factorInvolved($f$)$\vee$ checkSAT($testCase_{count}$)
12b:         increment $tries$
13:        insertValueForFactor($l$,$f$,$testCase_{count}$)
13a:       $mustSet$=mineMustAssignments()
13b:       for($l, f$) $\in mustSet$ do
13c:         insertValueForFactor($l$,$f$,$testCase_{count}$)
13d:         $p = p - f$
14:      saveCandidate$TestCasePool$,$testCase_{count}$)
15:    selectBestCandidate($testCasePool$)
16:    update($uncovered\text{-}t\text{-}set\text{-}count$)
17:    increment $numTests$
```

Algorithm 3: AETG-History Algorithm

consist of five models derived from four case studies of real software systems and thirty synthesized data sets that mimic the characteristics of the case study data.

4.1. Case Studies

In [6] we present case studies of two highly configurable, non-trivial software systems. These are the SPIN model checker[14] and the GCC compiler [11]. These studies result in three different models for testing configurable software. The first system, the SPIN Model checker, consists of two separate modules, a *simulator* (Spin$_s$) and a *verifier* (Spin$_v$). A single option turns on/off all features of these two modules, therefore a natural and conservative testing model would be to view these as two independent systems. The second case study is the GCC compiler. For this system we examined only a portion of the options, those related to the optimizer.

In this paper we present two additional case studies, Apache HTTP Server 2.2 [1] and Bugzilla 2.22.2 [20]. We describe each of these next.

Apache HTTP Server 2.2: The Apache HTTP Server 2.2 (Apa), is an open source web server. It can be customized by the system administrator through *directives*. The directives for Apache fall into nine categories, which include the core program, extensions, server config,etc. In total there are 379 configurable options. For the purposes of our case study we initially limited our examination to the 166 options related to *h* directives from the user manual. Upon further examination, we found that several of the constraints on this set of options involved an additional 6 factors that were not part of the *h* directives. We added those options to our model for a total of 172 options of which 92% are binary. We found 7 constraints in the Apache documentation that related between 2 and 5 different options. In total, only 18 options (or 10.5%) were involved in the 7 constraints.

Bugzilla 2.22.2: Bugzilla [20] (Bugz) is a defect tracking system from Mozilla. It maintains a database, generates reports, and includes time tracking features. In this system we again modeled a subset of the features. We used the ones that are found in the following sections of the documentation: Chapter 3. – *Administering Bugzilla*, Chapter 5. – *Using Bugzilla* and Chapter 6. – *Customizing Bugzilla*. This resulted in 44 factors. Once again we had to add additional factors, 10 in this case, to include all of those involved in the constraints. Our final model has 52 factors of which 94.2 % are binary. Bugzilla's documentation describes 5 constraints; 4 relating 2 options and 1 relating 3 options. In total, 11 options (or 21.2%) were involved in the 5 constraints.

In Table 1 we provide a summary of the covering array models for the 5 case studies that highlights their main characteristics. This table provides the number of factors for each as well as the break out of the number and percentage of the factors that have a specific number of values. It also presents the total number of constraints as well as the number and percentage of factors involved in constraints. It then gives the number and percent of the types of constraints by arity (i.e. 2-way, 3-way etc.). The last two columns in this table provide the number of constraints that individual factors participate in. We see this as an indication of constraint "coupling". For instance if a factor is involved in only a single constraint it will fall into the first category. We do not show data for factors involved in more than 2 constraints due to space limitations. For each of these systems, we have also enumerated the factors for the CAModel in an abbreviated form that shows the numbers of factors with a given value ($v^{\#f}$) in Table 2 as well as the arity of constraints ($arity^{\#constraints}$).

4.2. Simulated Data

We used our summarization of case study characteristics to synthesize 30 random covering array models with constraints that share the characteristics of the case study systems. Our synthesis algorithm starts by randomly generating a number of factors between 18 and 199 – the range of factors found in our case studies. It then randomly selects between 85-95% of the number of factors to be binary and the rest to involve between 3 and 6 factors. We weight this

	Factors and Values					Constraints					Factor Involv.	
	Num Factor	2 Values	3 Values	4 Values	5 or 6 Values	Num Cons	Factor Invol.	2-way Cons	3-way Cons	4 or 5-way Cons	1 Con. Per Factor	2 Con. Per Factor
Spin$_s$	**18**	13 (72.2)	0 (0.0)	5 (27.8)	0 (0.0)	**13**	9 (50.0)	13 (100.0)	0 (0.0)	0 (0.0)	5 (55.6)	0 (0.0)
Spin$_v$	**55**	42 (76.4)	2 (3.6)	11 (20.0)	0 (0.0)	**49**	33 (60.0)	47 (95.9)	2 (4.1)	0 (0.0)	12 (36.4)	7 (21.2)
GCC	**199**	189 (95.0)	10 (5.0)	0 (0.0)	0 (0.0)	**40**	36 (18.1)	37 (92.5)	3 (7.5)	0 (0.0)	14 (38.9)	13 (36.1)
Apa	**172**	158 (91.9)	7 (4.1)	4 (2.3)	2 (1.2)	**7**	18 (10.5)	3 (37.5)	1 (12.5)	3 (37.5)	14 (77.8)	1 (5.6)
Bugz	**52**	49 (94.2)	1 (1.9)	2 (3.8)	0 (0.0)	**5**	11 (21.2)	4 (80.0)	1 (20.0)	0 (0.0)	11 (100.0)	0 (0.0)

Table 1. Case Study Characteristics: Number and Percent of Factors/Constraints

decision with a 40% probability that 3 will be chosen, and a 20% probability for the rest. The percentage of constraints (in relation to the number of factors) in our systems varied from 4% to 89% and due to this large variation we chose to use the range in actual number of constraints, between 5 and 49, to synthesize constraints for our models. 80-100% of these are selected as binary constraints and the rest were chosen as 3, 4 or 5 way. We used a greedy generation approach, so at each point if we have assigned all constraints to a category we are done.

The other consideration that we tried to enforce is to make sure that between 40-100% of the factors involved in constraints are involved in only a single constraint while 10-20% of the factors are involved in two constraints. If there are any constraints that are not bound to factors, after this point, we randomly selected a factor to be involved with between 3 and 9 constraints.

Using this characterization, we generated 55 random samples. Of these, 21 were so highly constrained that they had zero feasible configurations. Our algorithms determined this in less than 2.5 seconds in every case. We selected the first 30 of the remaining 34 that produced valid CCIT samples and used those for our evaluation.

4.3. Technique Evaluation

We implemented Algorithm 2 (the base SAT algorithm) using two different SAT solvers. The first one, zChaff [17] is the same SAT solver that was used in [6]. This is written in C++ and does not use an incremental SAT solving approach – we label this **Basic SAT**. The second implementation of the basic sat algorithm uses MiniSAT-C v 1.14 [10], which implements an incremental SAT solving algorithm. We chose the C version because it was a small package that would be easy to comprehend for mining information – we label this **Inc SAT**. The implementation of Algorithm 3 also used MiniSAT. We added code to mine its internal data structures at each iteration of the algorithm to find must and may information – we label this **Hist SAT**. The last algorithm we used in our experiments is the base AETG implementation that does not handle constraints [4] – we label this **mAETG**. Although the covering arrays produced by this algorithm violate the constraints, we use this to show timing data for a completely unconstrained implementation of the algorithm.

We use the five test models from the case studies as well as our 30 synthesized models to determine how well each algorithm performs. This data is presented in Table 2. The count data, **No. SAT calls** and Covering Array **Size**, are averages over 50 runs rounded to the nearest whole number. We show the reduction in SAT calls (**% Dec.**) between Incremental and History SAT techniques since this is an important driver of CCIT algorithm cost. The timing data are the average of system and user time for 50 runs of each algorithm on an Opteron 252 processor running SUSE 10.1 rounded to the nearest tenth of a second. An initialization step occurs only once for all 50 runs and we amortize the cost of initialization across each of the runs. In addition to the individual runtimes, we show the percent increase (**% Inc.**) in the runtime of History SAT relative to the original mAETG algorithm (calculated on the full precision data); *negative* increases mean that the time actually decreased in the History SAT version. The aggregate data across the data set are shown in the **Avg** row.

4.4. Results Discussion

Our primary objective in this research was to investigate techniques for reducing the cost of CCIT sample generation without increasing sample size. The data from our evaluation clearly show the cost-reduction that can be achieved using incremental SAT solving and our history technique.

Incremental SAT yields, on average, a 19% speedup over the Basic SAT technique and History SAT yields an improvement of 33% over Basic SAT. More notably History SAT yields a 9.8% improvement over the unconstrained mAETG algorithm. This is remarkable because it is clear that History SAT performs a significant amount of additional work – on average more than 24 thousand SAT calls. The reduction in CCIT time is achieved because as the AETG algorithm moves across a row the must/may information mined from calculated SAT assignments is used to prune the space of choices left open to AETG in completing the row, thereby reducing AETG's search space.

A significant portion of this pruning effect is due to the use of must information which has the added benefit of reducing the number of SAT calls needed to calculate a sample. History SAT yields, on average, a 59% reduction in

Covering Array, t=2			No. SAT Calls				Time in Secs					Size	
	CAModel	**No. Cons.**	**Basic SAT**	**Inc SAT**	**Hist SAT**	**% Dec.**	**mAETG**	**Basic SAT**	**Inc SAT**	**Hist SAT**	**% Inc.**	**Basic SAT**	**Hist SAT**
Spin$_s$	$2^{13}4^{5}$	2^{13}	14,557	14,480	7,993	**44.8**	0.3	1.7	0.4	0.3	8.4	27	27
Spin$_v$	$2^{42}3^{2}4^{11}$	$2^{47}3^{2}$	97,379	95,848	37,021	**61.4**	8.2	32.2	11.3	8.5	3.8	43	43
GCC	$2^{189}3^{10}$	$2^{37}3^{3}$	57,388	55,432	31,089	**43.9**	217.6	320.0	286.9	204.0	**-6.2**	25	25
Apa	$2^{158}3^{8}4^{4}5^{1}6^{1}$	$2^{3}3^{1}4^{2}5^{1}$	44,199	40,088	32,687	**18.5**	278.7	318.6	249.2	244.1	**-12.4**	43	43
Bugz	$2^{49}3^{1}4^{2}$	$2^{4}3^{1}$	15,691	15,353	10,609	**30.9**	4.4	7.4	6.2	4.4	**-0.3**	25	25
1.	$2^{19}3^{1}5^{1}$	$2^{27}3^{5}4^{1}$	5,217	5,222	1,015	**80.6**	0.3	0.8	0.3	0.3	**-8.0**	15	15
2.	$2^{86}3^{3}4^{1}5^{5}6^{2}$	$2^{20}3^{3}4^{1}$	71,372	70,444	27,196	**61.4**	62.0	94.9	63.7	59.6	**-3.8**	56	56
3.	$2^{86}3^{3}4^{3}5^{1}6^{1}$	$2^{19}3^{3}$	49,351	49,034	19,784	**59.7**	41.3	59.7	53.5	38.1	**-7.7**	40	40
4.	$2^{27}4^{2}$	$2^{9}3^{1}$	8,265	8,271	1,974	**76.1**	0.9	1.8	1.1	0.8	**-5.4**	21	20
5.	$2^{51}3^{4}4^{2}5^{1}$	$2^{15}3^{2}$	21,626	21,390	9,740	**54.5**	7.5	12.5	9.9	6.7	**-11.2**	29	29
6.	$2^{155}3^{7}4^{3}5^{5}6^{4}$	$2^{32}3^{6}4^{1}$	138,852	137,133	73,590	**46.3**	400.1	531.6	520.2	373.6	**-6.6**	65	65
7.	$2^{73}4^{3}6^{1}$	$2^{26}3^{4}$	35,903	35,569	10,939	**69.2**	19.1	29.7	25.4	16.7	**-12.5**	34	34
8.	$2^{29}3^{1}$	$2^{13}3^{2}$	4,783	4,742	1,306	**72.5**	0.5	1.1	0.5	0.5	**-1.5**	12	12
9.	$2^{109}3^{2}4^{2}5^{3}6^{3}$	$2^{32}3^{4}4^{1}$	88,832	88,047	39,696	**54.9**	119.0	160.5	119.2	117.5	**-1.2**	57	57
10.	$2^{57}3^{1}4^{1}5^{1}6^{1}$	$2^{30}3^{7}$	24,477	24,302	5,874	**75.8**	10.8	14.5	8.3	7.8	**-27.8**	27	27
11.	$2^{130}3^{6}4^{5}5^{2}6^{4}$	$2^{40}3^{7}$	125,611	125,000	47,188	**62.2**	241.9	330.0	242.4	212.8	**-12.0**	64	64
12.	$2^{134}3^{7}5^{3}$	$2^{19}3^{3}$	73,425	72,355	38,018	**47.5**	151.4	193.7	187.5	131.8	**-12.9**	42	42
13.	$2^{84}3^{4}4^{2}5^{2}6^{4}$	$2^{28}3^{4}$	77,930	77,214	32,771	**57.6**	68.2	101.1	88.5	61.8	**-9.3**	61	60
14.	$2^{136}3^{4}4^{3}5^{1}6^{3}$	$2^{23}3^{4}$	104,256	103,440	58,480	**43.5**	216.1	283.2	212.8	201.1	**-6.9**	57	59
15.	$2^{124}3^{4}4^{1}5^{2}6^{2}$	$2^{22}3^{4}$	83,465	82,391	47,746	**42.0**	140.8	188.1	184.2	130.8	**-7.2**	51	51
16.	$2^{81}3^{5}4^{3}6^{3}$	$2^{13}3^{2}$	61,503	61,165	41,683	**31.9**	55.4	76.2	55.1	51.4	**-7.2**	56	56
17.	$2^{50}3^{4}4^{1}5^{2}6^{1}$	$2^{20}3^{2}$	31,367	31,416	14,231	**54.7**	10.6	17.7	11.2	9.9	**-6.3**	40	40
18.	$2^{110}3^{2}5^{2}6^{1}$	$2^{23}3^{3}4^{1}$	59,784	59,432	27,899	**53.1**	72.6	101.8	76.4	69.2	**-4.6**	41	42
19.	$2^{52}3^{3}4^{1}5^{1}$	$2^{21}3^{4}$	22,051	21,726	6,629	**69.5**	6.6	11.3	7.0	6.0	**-8.7**	28	28
20.	$2^{117}3^{5}4^{2}5^{5}6^{4}$	$2^{32}3^{7}$	119,054	117,566	42,021	**64.3**	178.9	257.8	190.0	179.5	0.3	65	65
21.	$2^{113}3^{7}4^{2}5^{3}6^{2}$	$2^{33}3^{5}$	90,480	90,018	37,725	**58.1**	132.8	186.8	131.8	118.1	**-11.1**	53	54
22.	$2^{64}3^{3}4^{2}5^{2}6^{1}$	$2^{37}3^{7}$	43,030	42,509	10,682	**74.9**	19.7	32.8	19.7	16.7	**-15.4**	40	40
23.	$2^{93}3^{6}4^{1}5^{1}6^{1}$	$2^{12}3^{2}$	48,968	48,204	30,218	**37.3**	49.3	72.8	66.3	47.2	**-4.2**	40	39
24.	$2^{78}3^{2}4^{5}5^{1}6^{4}$	$2^{27}3^{5}4^{1}$	74,293	73,146	19,975	**72.7**	56.3	85.9	73.2	49.9	**-11.4**	61	60
25.	$2^{72}3^{2}4^{1}5^{1}6^{2}$	$2^{33}3^{5}$	41,589	41,320	8,445	**79.6**	30.0	38.2	24.7	21.3	**-29.0**	40	40
26.	$2^{139}3^{2}4^{5}5^{5}6^{4}$	$2^{40}3^{7}$	127,108	124,191	55,057	**55.7**	294.1	387.4	279.1	252.4	**-14.2**	64	63
27.	$2^{39}3^{2}6^{2}$	$2^{18}3^{3}$	31,239	31,438	3,526	**88.8**	5.4	10.1	5.7	4.5	**-16.2**	48	48
28.	$2^{67}3^{4}5^{1}6^{1}$	$2^{32}3^{6}4^{1}$	32,168	31,524	7,351	**76.7**	18.5	26.1	15.7	14.5	**-21.7**	32	32
29.	$2^{29}4^{1}5^{1}$	$2^{26}3^{4}4^{1}5^{1}$	7,892	7,897	1,065	**86.5**	1.2	1.8	0.9	0.6	**-49.5**	16	16
30.	$2^{30}4^{1}6^{2}$	2^{5}	19,634	19,723	8,049	**59.2**	2.5	4.8	2.8	2.4	**-4.4**	46	46
Avg			55,793	55,058	24,265	**59.0**	83.5	114.1	92.3	76.1	**-9.8**		

Table 2. Average Number of SAT Calls and Time over 50 Runs

SAT calls relative to Incremental (or Basic) SAT. Furthermore, while not shown in the table, the number of SAT calls that yield an unsatisfiable result is over 21.6% for non-History SAT methods, but only 6.5% for History SAT. Since unsatisfiable SAT results are discarded, this means that History SAT saves on average over 8250 *useless* SAT calls for each CCIT sample calculation.

Coupling SAT and AETG clearly has performance benefits, but we were concerned that the heuristics used in each search might conflict leading to sub-optimal solutions. In practice, this does not appear to be a problem since the sum of the covering array sizes across all 35 problems for Basic and History SAT differ by one, 1462 and 1463 respectively.

We noted quite a bit of variation in the size and complexity of the synthesized configuration models. To get a sense of the scalability of our techniques on larger examples we analyzed the 10 examples with the largest number of factors: GCC, Apa, 6, 11, 12, 14, 15, 20, 21, and 26. For these 10 systems, the average reduction for History SAT was 9.1% relative to mAETG, which is in line with the overall data. Moreover, for these systems this reduction translates to nearly 16 fewer minutes to generate a CCIT sample. Greedy CIT algorithms are usually run multiple times to produce high-quality solutions, e.g., [2, 4] use between 50 and 20000 repetitions. In such a setting, our reductions for a single CCIT sample calculation would translate into a savings of between 13 hours and 219 days.

5. Conclusions

The conventional wisdom in the CIT community is that constraints significantly complicate the problem of computing a CIT sample. None of the existing techniques can deal with large numbers of interacting constraints without significantly increasing run-time or burdening the user [6]. The developers of the original AETG state that "Relations can also be highly constrained. Large numbers of constraints may significantly increase the amount of time required to find a solution." [16].

We believe that the techniques presented in this paper represent a significant advance in calculating CIT samples in the presence of constraints. In effect, synergistic integration of constraint satisfaction algorithms with CIT generation algorithms allows each algorithm to boost the performance of the other. The result is that high-quality constrained CIT samples can be produced for the same cost as unconstrained samples.

Our analysis of the 10 most complex examples in our data set is promising evidence of the benefits of our methods on realistically sized systems. Those systems, with an average of 153 factors, are several orders of magnitude smaller than extremely large-scale highly-configurable systems being developed in industry. Furthermore, while our study considered only pair-wise CCIT it is likely that for mission-critical systems engineers will target higher-order coverage which will dramatically increase the cost of CIT. More study is needed to better understand the scalability of our CCIT methods to extremely large-scale highly-configurable mission-critical systems.

6. Acknowledgments

This work was supported in part by the Army Research Office through DURIP award W91NF-04-1-0104, and by the National Science Foundation through awards 0429149, 0444167, 0454203, and 0541263.

References

[1] Apache Software Foundation. Apache HTTP sever. `http://httpd.apache.org/docs/2.2/mod/quickreference.html`, 2007.

[2] R. C. Bryce, C. J. Colbourn, and M. B. Cohen. A framework of greedy methods for constructing interaction test suites. In *Proceedings of the International Conference on Software Engineering*, pages 146–155, May 2005.

[3] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.

[4] M. B. Cohen, C. J. Colbourn, P. B. Gibbons, and W. B. Mugridge. Constructing test suites for interaction testing. In *Proceedings of the International Conference on Software Engineering*, pages 38–48, May 2003.

[5] M. B. Cohen, M. B. Dwyer, and J.Shi. Coverage and adequacy in software product line testing. In *Proceedings of the Workshop on the Role of Architecture for Testing and Analysis*, pages 53–63, July 2006.

[6] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *International Symposium on Software Testing and Analysis*, July 2007. to appear.

[7] C. J. Colbourn, M. B. Cohen, and R. C. Turban. A deterministic density algorithm for pairwise interaction coverage. In *IASTED Proceedings of the International Conference on Software Engineering*, pages 345–352, February 2004.

[8] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[9] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino. Applying design of experiments to software testing. In *Proceedings of the International Conference on Software Engineering*, pages 205–215, 1997.

[10] N. Eén and N. Sörrenson. MiniSAT-C v1.14.1. `http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/MiniSat.html`, 2007.

[11] Free Software Foundation. GNU 4.1.1 manpages. `http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/`, 2005.

[12] A. Hartman. Software and hardware testing using combinatorial covering suites. In *Graph Theory, Combinatorics and Algorithms: Interdisciplinary Applications*, pages 327–266, 2005.

[13] B. Hnich, S. Prestwich, E. Selensky, and B. Smith. Constraint models for the covering test problem. *Constraints*, 11:199–219, 2006.

[14] G. J. Holzmann. On-the-fly, LTL model checking with SPIN: Man pages. `http://spinroot.com/spin/Man/index.html`, 2006.

[15] D. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30(6):418–421, 2004.

[16] C. Lott, A. Jain, and S. Dalal. Modeling requirements for combinatorial software testing. In *Workshop on Advances in Model-based Software Testing*, pages 1–7, May 2005.

[17] S. Malik. zchaff. `http://www.princeton.edu/~chaff/zchaff.html`, 2004.

[18] J. P. Marques-Silva and K. A. Sakallah. GRASP: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[19] J. D. McGregor. Testing a software product line. Technical report, Carnegie Mellon, Software Engineering Institute, December 2001.

[20] Mozilla Organization. Bugzilla. `http://www.bugzilla.org/docs/tip/html/`, 2007.

[21] K. C. Tai and Y. Lei. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering*, 28(1):109–111, 2002.

[22] Y. Tung and W. S. Aldiwan. Automating test case generation for the new generation mission software system. In *Proceedings of the IEEE Aerospace Conference*, pages 431–437, 2000.

[23] A. W. Williams and R. L. Probert. A measure for component interaction test coverage. In *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications*, pages 301–311, Los Alamitos, CA, October 2001. IEEE.

[24] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, 31(1):20–34, 2006.