Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach

Myra B. Cohen, *Member*, *IEEE*, Matthew B. Dwyer, *Member*, *IEEE Computer Society*, and Jiangfan Shi, *Student Member*, *IEEE*

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. Many CIT algorithms lack a mechanism to avoid these. In recent work, automated constraint solving methods have been combined with search-based CIT construction methods to address the constraint problem with promising results. However, these techniques can incur a nontrivial overhead. In this paper, we build upon our previous work to develop a family of greedy CIT sample generation algorithms that exploit calculations made by modern Boolean satisfiability (SAT) solvers to prune the search space of the CIT problem. We perform a comparative evaluation of the cost effectiveness of these algorithms on four real-world highly-configurable software systems and on a population of synthetic examples that share the characteristics of those systems. In combination, our techniques reduce the cost of CIT in the presence of constraints to 30 percent of the cost of widely used unconstrained CIT methods without sacrificing the quality of the solutions.

Index Terms—Combinatorial interaction testing, constraints, covering arrays, propositional logic, satisfiability (SAT) checking.

1 Introduction

Software development is shifting from the production of individual programs to the production of families of related programs [1]. This eases the design and implementation of multiple software systems that share a common core set of capabilities, but have key differences, such as the hardware platform they require, the interfaces they expose, or the optional capabilities they provide to users. Often times, significant reuse can be achieved by implementing a set of these systems as one integrated *highly-configurable* software system. Configuration is the process of binding the optional *features* of a system to realizations in order to produce a specific software system, i.e., a member of the family.

The concept of a highly-configurable software system arises in many settings differentiated by the point in the development process when feature binding occurs. An example of very early feature binding is seen in software product lines (SPLs). An SPL uses an architectural model to define a family of products built from a core set of platforms and customized through the identification of points of variability and commonality. Variability points allow the developer to plug in different variants of features while still maintaining the overall system architecture. At the other end of the spectrum are dynamically reconfigurable systems, where feature binding happens at runtime and may happen repeatedly. NASA's Deep Space 1 Remote Agent software is an example of such a system that uses online planning to activate and deactivate modules in the system based on spacecraft and mission status [2]. In between are the common class of user configurable systems. These are programs such as desktop applications, web servers, or databases that allow users to modify a set of predefined options, e.g., command-line parameters or configuration file settings, as they see fit and then run the program with those options.

Highly-configurable systems present significant challenges for validation. The problem of testing a single software system has been replaced with the problem of testing the set of software systems that can be produced by all of the different possible bindings of optional features. A single test case may run without failing under one configuration, however, the same test case may fail under a different configuration [3], [4]. One cause for this is the unintended *interaction* of two or more bindings for features.

Fig. 1 presents a simplified mobile phone product line that we use to illustrate the challenges of testing highly-configurable software; the challenges are present in systems with later binding times as well. This product line is hypothetical, but its structure is reflective of portions of the Nokia 6000 series phones [5]. The product line supports three display options (16MC, 8MC, and BW) and can have either a text email viewer (TV), a graphical email viewer (GV), or no viewer (NOV). A phone may be built with two types of camera (2MP or 1MP) or without a camera (NOC), with a video camera (VC) or without a video camera, and with support for video ringtones (VR) or without that support. There are a total of 108 $(3 \times 3 \times 3 \times 2 \times 2)$ different

• *The authors are with the Department of Computer Science and Engineering, Avery Hall, University of Nebraska-Lincoln, Lincoln, NE 68588-0115. E-mail: {myra, dwyer, jfshi}@cse.unl.edu.*

Manuscript received 29 Oct. 2007; revised 28 Feb. 2008; accepted 12 May 2008; published online 27 June 2008.
Recommended for acceptance by S. Elbaum and D.S. Rosenblum.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2007-10-0305.
Digital Object Identifier no. 10.1109/TSE.2008.50.

0098-5589/08/$25.00 © 2008 IEEE Published by the IEEE Computer Society

Possible Values	Product Line Options (factors)				
	Display	**Email Viewer**	**Camera**	**Video Camera**	**Video Ringtones**
	16 Million Colors ***16MC***	Graphical ***GV***	2 Megapixels ***2MP***	Yes ***VC***	Yes ***VR***
	8 Million Colors ***8MC***	Text ***TV***	1 Megapixel ***1MP***	No ***!VC***	No ***!VR***
	Black and White ***BW***	None ***NOV***	None ***NOC***		

Constraints on Valid Configurations:

(1) *Graphical email viewer* **requires** *color display*
(2) 2 Megapixel camera **requires** a *color display*
(3) *Graphical email viewer* **not supported** with the *2 Megapixel camera*
(4) 8 Million color display **does not support** a 2 Megapixel camera
(5) *Video camera* **requires** a *camera* **and** a *color display*
(6) *Video ringtones* **cannot occur** with *No video camera*
(7) The combination of *16 Million colors*, *Text* and *2 Megapixel camera* will **not be supported**

Fig. 1. Mobile phone product line.

phones that can be produced by *instantiating* this SPL. For each one, we will need to run a test suite if we wish to fully test the family of products.

A test run on two different software products may behave differently even if those products only differ in features that are seemingly unrelated to the test. For instance, a problem with the email viewer may not appear under the 8MC display, but may only appear with 16MC because of its increased memory usage.

For large SPLs, testing the complete set of SPL instances is impractical, however, testing techniques that sample that set can be used to find interaction faults. A common approach to such testing is to systematically sample the set of instances in such a way that all t-way combinations of features are included; pairwise or two-way combinations are the most commonly studied. This is commonly called *combinatorial interaction testing* (CIT) [6]. A significant literature exists that describes foundational concepts, algorithms, tools, and experience in applying interaction testing tools to real systems [4], [6], [7], [8], [9], [10], [11], [12]. One prime objective of this body of work is to produce the smallest subset of configurations for a system that achieves the desired t-way coverage [6], [8], [9], [13], [14], [15], [16], [17], [18], [19].

In most configurable systems, *constraints* or dependencies exist between features. At the bottom of Fig. 1, we list seven constraints that have been placed on the valid product instances. Constraints may arise due to any number of reasons, for example, inconsistencies between certain hardware components, limitations due to available memory and software size, or simply marketing decisions. We present natural language representations of constraints since that is how they are typically described in software documentation. Taken together, these constraints reduce the number of product instances to 31, but, rather than simplifying CIT methods, constraints present significant challenges for cost-effective CIT.

Constraints were first described as being important to CIT in [6], but, rather than applying algorithmic techniques, remodeling of the program input is required. This may result in slightly different test models and requires manual intervention. A similar approach is seen in [20]. Much of the literature simply ignores constraints [8], [13], [16], [17], [18], [19], but, as we show, this leads to CIT solutions in which the overwhelming majority of the calculated configurations violate the constraints and are thus useless for driving the testing process. Some researchers have suggested approaches that require the user to explicitly define all illegal configurations for processing by their algorithm [14], [21], but, as we show, even a small number of constraints can give rise to enormous numbers of illegal configurations and asking a user to produce those is intractable. Researchers have attempted to bias CIT methods to avoid constraints [22], but, as we show, those methods are guaranteed to produce illegal configurations in the relatively common situation where multiple constraints interact to produce additional *implicit* constraints. Finally, researchers have concluded that the treatment of constraints is a straightforward "engineering extension" that is, presumably, not worthy of investigation, but, as we show, the development of cost-effective CIT methods that treat constraints is nontrivial.

In this paper, we build on our initial work on this subject. We identified and quantified the fundamental challenges of treating constraints in CIT methods and developed an algorithmic framework for incorporating constraints into existing greedy and simulated annealing algorithms for interaction test generation using an off-the-shelf satisfiability (SAT) solver [23]. Follow-on work proposed an optimization for our greedy algorithm that more tightly integrated intermediate calculations produced by the SAT solver to significantly reduce the cost of CIT without negatively impacting the quality of solutions [24]. This paper

1. provides an integrated presentation of our previous constraint-aware greedy CIT methods,
2. develops an additional optimization to reduce solution generation time significantly over the approach in [24],
3. quantifies the number and complexity of constraints and the problem with ignoring them in CIT using a set of five real-world highly-configurable software case studies on four software subjects, and, finally,
4. presents a comparative evaluation of our previous and new CIT methods on those case studies and on 30 synthesized configurable system models that mimic the characteristics of the case studies.

Our results demonstrate that high-quality interaction test suites for systems that are highly constrained can be generated using our carefully engineered CIT methods in less time than in methods that ignore constraints.

The remainder of this paper is organized as follows: In Section 2, we provide background on interaction testing, satisfiability checking, and our previous work. Section 3 discusses the need for constraint support in CIT algorithms. Section 4 presents a series of four greedy CIT algorithms that explicitly treat constraints as variations on a common core algorithm; this allows the differences between algorithms to be clearly illustrated and justified. Section 5 presents our case studies, the methodology for synthesizing configurable system models, and the results of applying our algorithms to the case studies and a collection of synthesized models. Section 6 discusses related work in more depth and Section 7 concludes and discusses future work.

2 Background

In the example in Fig. 1, the mobile phone SPL contains a total of 108 possible product line instances before incorporating

	Display	Email Viewer	Camera	Video Camera	Video Ringtones
1*	16 Million Colors	None	1 Megapixel	Yes	Yes
2*	8 Million Colors	Text	None	No	No
3	16 Million Colors	Text	2 Megapixels	No	Yes
4	Black and White	None	None	No	Yes
5	8 Million Colors	None	2 Megapixels	Yes	No
6	16 Million Colors	Graphical	None	Yes	No
7	Black and White	Text	1 Megapixel	Yes	No
8	8 Million Colors	Graphical	1 Megapixel	No	Yes
9	Black and White	Graphical	2 Megapixels	Yes	Yes

Fig. 2. Pairwise CIT sample ignoring constraints.

system constraints. The number of possible instances of a product line grows exponentially in relation to the number of feature variants. If there are four features, each with five possible variants, there are 5^4 possible product instantiations. Developers may wish to generate a set of configurations to perform testing of the whole product line. Testing all possible instances of the product line, however, is usually intractable; therefore, as a validation method, this will not scale. One method that can be used is to systematically sample all pairs or t-way combinations of feature variants [6]. We call this *CIT sampling*.[1]

Fig. 2 is an example of a pairwise or two-way CIT sample for the phone SPL ignoring constraints; we will consider them in the following sections. In this example, we have a set of *configurations* (a product instance that has one variant selected for each feature) that combines all pairs of variants between any two of the features. For instance, all displays are combined with both of the email viewers as well as without any email viewer.

Many studies have shown that CIT is a powerful sampling technique for functional input testing that may increase the ability to find certain types of faults efficiently [6], [7], [26] and may provide good code coverage [6], [11]. Recent work on CIT has studied its use on both user configurable systems [3], [4] and on SPLs [25], [27] and has shown that CIT performs more consistently than random sampling [4].

A primary focus in the literature on interaction testing has been on developing new algorithms to find smaller t-way samples [9], [14], [15], [17], [19], [28]. However, much of this literature ignores practical aspects of applying CIT to real systems, which limits the effectiveness and applicability of this work. In this paper, we focus on one difficult, yet prevalent, issue which may confound existing algorithms—the handling of constraints.

2.1 CIT Samples: Covering Arrays

Before we discuss the various techniques for constructing CIT samples, we begin with some definitions. A t-way CIT sample is a mathematical structure, called a covering array.

Definition 2.1. *A* covering array $CA(N; t, k, |v|)$ *is an* $N \times k$ *array from a set* v *of symbols with the property that every* $N \times t$ *subarray contains all ordered subsets of size* t *from the* $|v|$ *symbols at least once.*

The *strength* of a covering array is t, which defines, for example, two-way or three-way sampling. The k columns of this array are called *factors*, where each factor has $|v|$ values. Although the trivial mathematical lowest bound for a covering array is $|v|^t$, this is often not achievable and sometimes the real bound is unknown [14].

As seen in the phone product line, most software systems do not have the same number of values for each factor, i.e., we do not have a single $|v|$. A more general structure can be defined that allows for variability in factor-value domain size.

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

We use a shorthand notation to describe mixed-level covering arrays by combining entries with equally sized value ranges, i.e., $|v_i| = |v_j|$, where $i \neq j \wedge 1 \leq i, j \leq k$. For example, three factors that can each take on two values can be written as 2^3. We drop the explicit use of k from our model since this can be inferred by adding the superscripts. Fig. 2 illustrates a two-way CIT sample, $MCA(9; 2, 3^3 2^2)$, for the phone example. There are five factors in this model, three of which have three values, while the other two are binary. In this paper, when we use the term *covering array* we will use it to mean both standard and mixed-level arrays as appropriate.

2.2 Finding CIT Samples

Many algorithms and tools exist that construct covering arrays. For certain values of t, k, $|v|$ mathematical techniques (both direct and recursive) can be used to construct covering arrays [14], [19], [28]. Although constructions produce small covering arrays efficiently, they are not general. Several examples of metaheuristic search have appeared in the literature, such as simulated annealing, genetic algorithms, and tabu search [9], [16]. We do not consider this class of techniques in detail in this paper, but note that, in general, they tend to produce smaller covering arrays at increased cost relative to greedy methods when constraints are considered [23].

There are two primary classes of greedy algorithms that have been used to construct covering arrays. The majority of algorithms are the one-row-at-a-time variation of the automatic efficient test case generator (AETG) [6]. This type of algorithm is the focus of this paper and is described in more detail in Section 4. A different type of greedy algorithm is the In Parameter Order (IPO) algorithm [17]. Rather than focusing on a row-at-a-time, it generates all t-sets for the first t factors and then incrementally expands the solution, both horizontally and vertically, using heuristics until the sample is complete.

The presence of constraints demands a new definition for a proper CIT sample. Constraints may, for example, disallow combinations of options or require that when one option value is selected that another also be selected. Clearly, a

1. The work on applying CIT to software product lines is still emerging (see [25] for open challenges), but this example is representative of other types of configurable software on which CIT has been shown to be effective [4].

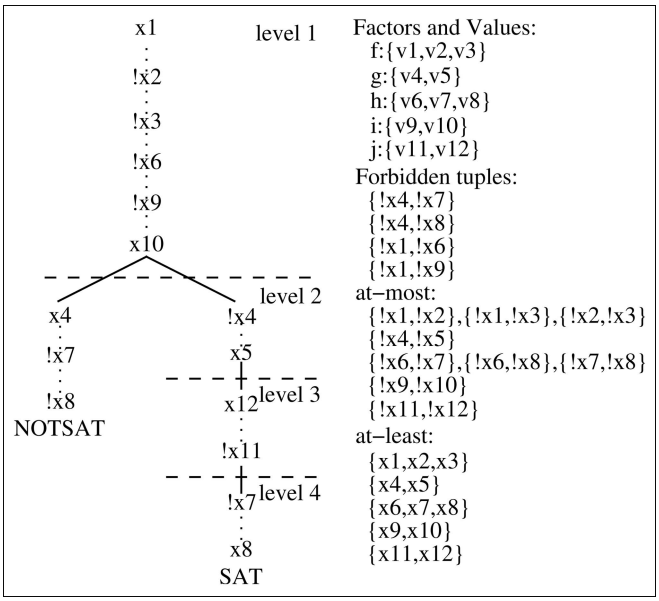

Fig. 3. Example SAT search.

configuration generated by a CIT method that does not consider such constraints may be inconsistent with those constraints and thus is not a valid system configuration. Examining the first constraint presented in Fig. 1, we see that a valid system cannot have both a black and white display and a graphical email viewer. We refer to such a combination as a *forbidden tuple*; Section 2.4 defines this concept precisely. In our original CIT sample, Fig. 2, we see that this constraint has been violated by the last configuration.

2.3 Boolean Satisfiability Solving

In Section 4, we will present a synergistic integration of AETG and Boolean satisfiability solving algorithms. To understand this integration, a basic understanding of modern SAT solving algorithms is required.

Most practical 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 [29] 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 [30].

Fig. 3 illustrates the data structures built for a satisfiability check of a partial row in a covering array for three binary, g, i, and j, and two ternary, f and h, factors. Each factor-value is assigned a unique value. For instance, factor f has three values, $v1$, $v2$, and $v3$. 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 constraints in this CIT problem consist of four forbidden tuples that restrict the values that can be bound to those factors. For example, the first two forbidden tuples require that $h = v6$ if $g = v4$. Two other types of constraints needed for this problem are shown. These are *at-least* and *at-most* constraints. These are needed to ensure that there are no empty factors included in our solution (at-least) and that we can assign only one value to a factor at a time (at-most). The figure shows two branches of a search for a satisfying assignment given an input clause $x1$, i.e., $f = v1$.

SAT solvers divide the process of finding a satisfiable assignment 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. A propagation phase (denoted by a consecutive run of 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. Note that the order in which variables and truth assignments are chosen in the search phase and the order in which clauses are considered in the propagation phase may be randomized or controlled by heuristics that are specific to a given SAT solver implementation. Consequently, the example search in Fig. 3 illustrates one possible SAT search.

The BCP phase attempts to produce *unit clauses* in order to infer a truth assignment for a variable; a unit clause 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 clause implies its truth assignment. In Fig. 3, the first step is an implicit search step reflecting the initial assignment of $x1$ to true. The next five 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 $x10$ is assigned, the algorithm performs a search step where it chooses to explore assignments with $x4$ in the left branch; each search step increases the *level* of the search. A second BCP phase follows where $!x7$ is inferred from the clause $\{!x4, !x7\}$, after which $!x8$ is inferred. 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. The truth assignments $x1$, $x5$, $x8$, $x10$, and $x12$ correspond to a covering array row in which $f = v1$, $g = v5$, $h = v8$, $i = v10$, and $j = v12$.

In general, when the search backtracks, some subset of the truth assignment is responsible for a conflict among clauses that leads to the failure to find a satisfying assignment, denoted NOTSAT. In the example in Fig. 3, the combination of $x4$ and $!x6$ will always lead to a conflict with the clause $\{x6, x7, x8\}$. Conflict-clause learning techniques in SAT solvers perform a dependence analysis of the sequence of truth assignments and the clauses that influenced those assignments 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 is 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. Once detected, this clause will be present in all subsequent SAT solver calls.

2.4 Constrained CIT

Our previous work developed several approaches to incorporating constraints in CIT. The presentation here refines the approach in [24] and supersedes the approach in [23]. We model constraints as Boolean formulae defined over atomic propositions that encode factor-value pairs; Fig. 1 informally introduced such propositions. We focus here on how constraints expressed in a canonical form of Boolean formulae are integrated into CIT problems; in [24], we discuss the translation of different forms of natural language constraints into that canonical form.

There are many different ways to encode a constraint problem for SAT. We use a direct encoding for constrained CIT (CCIT) problems. In certain applications, alternative encodings, such as log encodings, may provide performance benefits for SAT [31]. Typically, one considers such alternative encodings when the translation into SAT is particularly expensive, e.g., it suffers exponential blowup, or when the SAT solving algorithm is optimized for a particular encoding. In our work, we have found that translating a CCIT problem to SAT is fast, it avoids exponential blowup, and it results in encodings that can be solved in less than 1 second per SAT call across all of our experiments.

Let $\{f_1, \ldots, f_k\}$ be the set of factors in a CIT problem and let P be defined as a set of propositions, one for each possible pair of factor, $f \in \{f_1, \ldots, f_k\}$, and value, $v \in v_f$, e.g., $f = v$, where v is the value assigned to f. In this paper, we use v_f to represent the set of values associated with a factor and $|v_f|$ to represent the size of that set. We denote the factor associated with a proposition $p \in P$ as $f(p)$.

We encode constraints as conjunctions of positive and negative occurrences of propositions from P that may never occur together in a row of a covering array.

Definition 2.3. *A* forbidden tuple *is a set of propositions, $q \subseteq P$, where each proposition defines a value for a distinct factor, $\forall p_i, p_j \in q, p_i \neq p_j : f(p_i) \neq f(p_j)$.*

Without loss of generality, a proposition may be replaced with its negation in a tuple; for simplicity in our presentation, we only use positive occurrences of propositions.

Definition 2.4. *A* forbidden tuple constraint *is defined as $\phi_q \equiv \neg \bigwedge_{p \in q} p$ and, if F is the set of forbidden tuples in the CCIT problem, then $\phi_F \equiv \bigwedge_{q \in F} \phi_q$.*

Note that the conjunction of a set of forbidden tuple constraints may give rise to additional forbidden tuples. We refer to these as *implicit* constraints [23] since they do not appear in the initial formulation of the CIT problem.

A covering array has a value for each factor in each row and this is encoded by *at-least* constraints in our model [15]. As illustrated in [23], at-least constraints are needed to ensure the discovery of implicit constraints.

Definition 2.5. *For each factor, $f \in \{f_1 \ldots f_k\}$, an* at-least constraint *is $\psi_{f \geq 1} \equiv \bigvee_{v \in v_f} f = v$. The set of all at-least constraints is $\psi_{\geq} \equiv \bigwedge_{f \in \{f_1, \ldots, f_k\}} \psi_{f \geq 1}$.*

An *at-most* constraint [15] ensures that at most one of the propositions binding a value to a factor is true.

Definition 2.6. *For each factor $f \in \{f_1 \ldots f_k\}$, an* at-most constraint *is $\psi_{f \leq 1} \equiv \bigwedge_{v, v' \in v_f \wedge v \neq v'} f \neq v \vee f \neq v'$. The set of all at-most constraints is $\psi_{\leq} \equiv \bigwedge_{f \in \{f_1, \ldots, f_k\}} \psi_{f \leq 1}$.*

Together, at-most and at-least constraints ensure that exactly one factor-value proposition is true for each factor in the CIT problem.

Definition 2.7. *For a CIT problem, $C = \phi_F \wedge \psi_{\geq} \wedge \psi_{\leq}$ are the set of* base constraints.

All configurations produced for a CCIT problem must satisfy the base constraints. We note that such a satisfiability test naturally accounts for implicit constraints, so they need not be included in C.

Definition 2.8. *Given a CIT problem with base constraints C, a given tuple encoded as a Boolean formula, s, is* C-consistent *if $\phi_F \wedge \psi_{\geq} \wedge \psi_{\leq} \wedge s$ is satisfiable.*

This definition is particularly convenient since it handles *partial configurations*. This permits its use in greedy algorithms, which must evaluate a configuration's consistency incrementally during the construction of each covering array row. We extend the definition of a CIT sample to enforce C-consistency.

Definition 2.9. *A* constrained mixed-level covering array, *$CMCA(N; t, k, (|v_1|, |v_2|, \ldots, |v_k|), C)$, 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 $N \times t$ subarray cover all C-consistent t-tuples of values from the t columns at least one time, and 3) all rows are C-consistent.*

3 The Need for Constraint Support

There has been relatively little discussion in the CIT literature of how to construct a covering array in the presence of constraints. Bryce and Colbourn demonstrate that determining whether a configuration exists that satisfies a given set of constraints is an NP-hard problem [22]. They also argue that there is a strong need for a workable solution. In [23], we survey existing CIT techniques, assess their support for constraints, and argue *qualitatively* that the support is lacking in several regards. In this section, we revisit and *quantify* the extent to which existing CIT techniques are insufficient for supporting constraints that arise in realistic CIT problems.

We summarize details of a quantitative analysis of five case studies introduced in [23], [24] and a collection of CIT problems that we synthesized to reflect the number and diversity of constraints in those case studies—the data are presented in detail in Section 5. Specifically, we present data on the impact of ignoring constraints and the prevalence of *implied forbidden t-sets* (forbidden tuples matching the arity of the strength of testing). As explained in [23], the coverage of all required t-sets in the CIT sample is the stopping criteria for most CIT algorithms, making the discovery of implicit t-sets necessary.

3.1 The Impact of Ignoring Constraints

It seems natural to ask if it is viable to generate covering arrays without regard to constraints, followed by a postprocessing pass to delete the configurations violating constraints and to regenerate new configurations that cover the t-sets that deleted configurations uniquely covered. Intuitively, this makes sense if the number of constraints and arity of constraints relative to the CIT problem is such that

a small proportion of configurations will need to be regenerated. Unfortunately, it is our experience that, for realistic CIT problems, a significant majority of the configurations will need to be regenerated. Consider the seven constraints at the bottom of Fig. 1. Violating any one constraint in a configuration will cause the configuration to be judged invalid and force its removal from the covering array. In Fig. 2, seven out of nine configurations violate some constraint—configurations that do not violate a constraint are starred. For example, configuration 3 violates constraint 7. The two valid configurations (1 and 2) cover only 35 percent of the legal two-way combinations for this system; 65 percent must be covered by newly generated configurations. While this is a simple example with a seemingly large ratio of constraints to factors, having only 78 percent of the MCA configurations violate constraints is actually better than the situation we found in our broader study.

Table 3 shows data for 35 CIT problems with constraints. The numbers are averages over 50 runs of each algorithm. The fourth column (labeled **mAETG size/SAT rows**) gives data on the number of configurations generated for each problem using the unconstrained AETG algorithm (**size**) and, separated by a "/", the number of those configurations that satisfied all constraints (**SAT rows**). On average, 96 percent of the configurations of an unconstrained sample violate at least one constraint. The minimum percentage of configurations violating constraints was 64 percent, while the median and mode were both 100 percent. This is strong evidence that constraint handling must be incorporated into CIT generation methods rather than added on as a postprocessing phase. The algorithms we present in Section 4 do precisely that.

3.2 The Prevalence of Implied Forbidden t-Sets

Consider a CIT problem that is targeting t-way coverage. Each forbidden tuple of arity t trivially implies a t-way combination that cannot be present anywhere in the CIT sample. More generally, the conjunction of a set of forbidden tuples of varying arity *imply* a number of t-sets, none of which can be present in the CIT sample.

These *forbidden t-sets* may not be obvious to the person modeling a highly-configurable system [12]. Consider constraints (5-6) from Fig. 1. The three forbidden pairs are: $\{!VC, !NOC\}$, $\{!VC, !BW\}$, and $\{!VR, !VC\}$, where we use value names for propositions. It is not difficult to prove that the conjunction of these clauses implies the clause $\{!VR, !BW\}$. In other words, there may be constraints on legal configurations of a CIT problem that are not stated explicitly—we call these implicit forbidden tuples.

Some researchers have proposed approaches that require users to explicitly state all forbidden configurations which will bypass the need to know about implicit constraints [21]. As we argue in [23], in the worst case, there may be combinatorially many invalid configurations and calculating them would be effectively infeasible for a user, but it is unclear how frequently implied constraints occur in practice. The data in our evaluation shed some light on this. Across the set of 35 realistic CIT problems we studied, there are a total of 840 explicit forbidden pairs, or two-sets, and 24,247 implicit forbidden pairs. This suggests that the worst case can appear in practice. It is important to note that there is significant diversity across the 35 problems we considered; three problems have no implicit forbidden pairs, five problems have fewer implicit than explicit forbidden pairs, and three problems have two orders of magnitude more implicit than explicit forbidden pairs.

The problem of implied forbidden t-sets is exacerbated as the strength of CIT coverage increases relative to the arity of constraints. When forbidden tuples of arity less than t are present, the number of implied forbidden t-sets will, in general, be exponential in the number of factors. Our data bear this out. Across the 35 CIT problems, the total of 25,087 forbidden pairs, explicit and implicit, and 122 explicit forbidden triples give rise to over 3 million implied forbidden triples. As in the case of pairs, the data show significant diversity, but even the problems with the smallest numbers of pairs give rise to many hundreds or thousands of implied forbidden triples.

Clearly, the prevalence of implied forbidden t-sets demands that they be accounted for in an automated fashion. The algorithms we present in Section 4 do precisely that.

4 Algorithms for CCIT

In this section, we present a series of AETG-like algorithms for computing CMCAs for CCIT problems. We begin with a basic integration of C-consistency checks, realized through calls to existing SAT solvers, into the AETG algorithm. We then discuss two optimizations of this algorithm that seek to more tightly integrate the AETG and SAT algorithms. These optimizations observe that the basic algorithm alternates phases in which AETG and SAT each search the space of possible assignments of values to factors in a configuration. While AETG and SAT operate on different encodings of a configuration and employ completely unrelated heuristics, they can produce information during their respective phases that can be used to reduce the cost of subsequent search phases. The first optimization exploits properties of the sequence of SAT solver calls to mine information about assignments to unbound factors that can optimize subsequent AETG search phases; this was first presented in [24]. We call this AETG-History. The second optimization is novel and involves an aggressive mining of variable assignments made by SAT after a threshold point has been reached in constructing a configuration. We call this AETG-Threshold. Finally, we combine these two optimizations in a way that leverages both of their strengths.

4.1 AETG with C-Consistency Checks

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 AETG [6]. Multiple variants of AETG have appeared in the literature, e.g., [8], [10], [13], [18]. 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 [6].

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

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=¬factorInvolved(f) ∨ 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=¬ factorInvolved(f) ∨ 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 randomly shuffled, line 10, and then the best value for each factor is selected, lines 12-13, where the best value produces the most previously uncovered t-sets. In each step, where a "best" decision is made, as well as where the first factor and value is selected in lines 7-9, ties are broken randomly, causing nondeterminism in differing runs of the algorithm. Other greedy algorithms [13], [18] use slightly different heuristics to select the factor ordering.

Algorithm 2 illustrates the integration of C-consistency checks into our AETG-like algorithm, which we first presented in [23]. In this algorithm, the *CAModel* has been extended to include constraints. The original AETG-like algorithm is modified in three areas. 1) We piggy back onto the initialization step (not shown) a calculation of the set of factors that is involved in at least one constraint, `factorInvolved`—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. 3) Consistency checks, lines 8a and 12a, are introduced to determine if there exists an extension of the partial row to a full row that 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).

4.1.1 Leveraging Incremental SAT Solving

Most modern SAT solvers are incremental in the sense that they use conflict-clause learning to infer additional clauses across a sequence of SAT problems. SAT problems arising in CCIT solutions have a natural two-level structure: 1) the sequence of calls within a row and 2) the sequence of rows. Conflict-clause learning based on the entire SAT formula can be performed within a row, but one cannot learn clauses that are dependent on the values within one row and then use those clauses to solve SAT problems for another row. For this reason, we use MiniSAT [32], which allows a set of clauses to be passed as *assumptions*. Its conflict-clause learning algorithm only stores clauses that are independent of assumption clauses. We define clauses for the base constraints and incorporate 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; in our current approach, we do not apply clause learning within a row. We use this incremental MiniSAT technique to implement the AETG-SAT algorithm, which serves as a baseline method in the evaluation presented in Section 5.

4.2 Exploiting SAT History

Clause-learning improves SAT performance based on CMCA row-independent information. We improve performance further by exploiting the similarity of SAT calls within a row. In Algorithm 2, 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 2.4, 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 jth iteration of the loop is $C \wedge f_1 = v_1 \wedge f_2 = v_2 \wedge \ldots \wedge f_j = v_j$, using individual distinct atomic propositions to encode each $f_i = v_i$. If the `checkSAT` call on line 12a succeeds, then, on the $(j+1)$st iteration of the loop, the formula $C \wedge f_1 = v_1 \wedge f_2 = v_2 \wedge \ldots \wedge f_j = v_j \wedge f_{j+1} = v_{j+1}$ will be checked for satisfiability. The description of SAT solving in Section 2.3 makes it clear that checking this formula for SAT will involve an initial, level 1, BCP phase that infers the variable assignments that are solely dependent on the given partial assignment and the base constraint clauses.

Given propositional formulae, p, q, and r, $(p \Rightarrow q) \Rightarrow ((p \wedge r) \Rightarrow q)$. In other words, if all satisfying truth assignments for p make q true, then all satisfying truth assignments for an extension of p, via conjunction, will also make q true. In our setting, p is comprised of the conjunction of base clauses and the factor-value assignment for the partial row at the point where the jth SAT search call is made, q is comprised of the conjunction of assignments inferred in the level 1 BCP phase of that search, and r is comprised of the conjunction of a set of factor-value assignments that will be submitted to a future SAT call made for the current row. Consequently, mining assignments from the level 1 BCP phase of a SAT search yields a

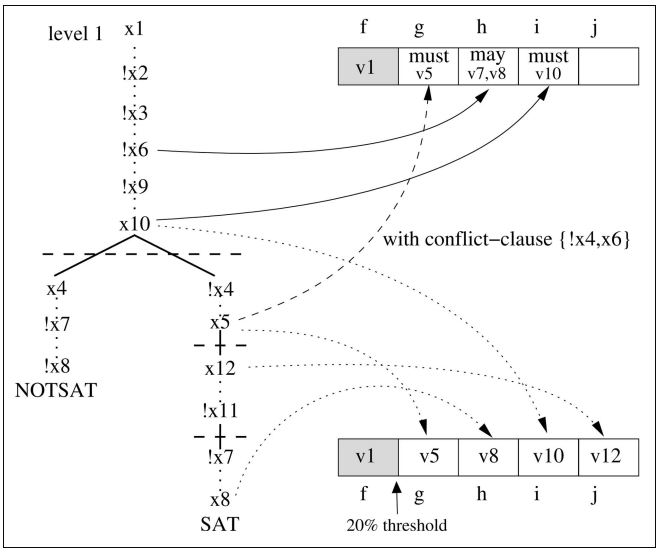

Fig. 4. Example of mining must/may assignments from a SAT search.

set of variable assignments that can be fed back to AETG to prune its search for a row.

There are two interesting cases. These cases are illustrated in Fig. 4, which shows how the result of the SAT search in Fig. 3 starting with factor-value binding $f = v1$ is used to prune future AETG processing:

1. When level 1 BCP infers a positive variable assignment, e.g., $x10$, this implies that all satisfying assignments for rows that extend the current partial assignment, e.g., $f = v1$, must have the designated factor-value binding, e.g., $i = v10$. We refer to this as *must* information for factor-value bindings. AETG exploits must information to skip the assignment of values to the factor later in a row, as described below.
2. When level 1 BCP infers a negative variable assignment, e.g., $!x6$, this implies that no satisfying assignment of a full configuration that extends the current partial assignment, e.g., $f = v1$, can possibly have the designated factor-value binding, e.g., $h = v6$. If we subtract such negated variables from the set of variables for that factor, the resulting set of variables is referred to as *may* information for factor-value bindings. AETG exploits may information to prune the set of possible values it chooses from for assignment to the factor later in a row as described below, e.g., for h, only $v7$ and $v8$ need be considered.

An additional subtlety arises in our approach. When conflict-clause learning determines that certain assignment combinations prevent a satisfying assignment in the context of a given partial row, e.g., $\{!x4, x6\}$ from the discussion of Fig. 3, this can introduce additional must or may information. Our implementation does not attempt to extract this information directly from the calculated conflict-clauses; rather, we let the conflict-clause learning mechanism assert those clauses for use in subsequent SAT calls for the given row. The effect of this is illustrated by the dashed arrow in Fig. 4, which indicates that a subsequent SAT call will exploit the conflict-clause to extend the level 1 BCP phase to infer $!x4$ and, subsequently, $x5$, which yields the must information $g = v5$.

Algorithm 3 illustrates the extensions to Algorithm 2 (AETG-SAT) with must/may optimizations—the new statements are shown in bold. We define two methods that mine the path in the SAT search tree corresponding to the SAT assignment. More specifically, the portion of that path corresponding to BCP level 1 is analyzed in:

- `mineMayAssignments` to return, for each factor, the possible value assignments that have not been eliminated by the existence of a *negative* occurrence of a proposition for a factor-value assignment, and
- `mineMustAssignments` to return the set of factor-value pairs that must be present due to the existence of a *positive* occurrence of a proposition for that factor-value assignment.

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 reduces the chance of selecting a value that will lead to an inconsistent partial configuration. Note that it is still possible for the AETG-History algorithm to produce unsatisfiable partial configurations. This is because the mined assignment information is based only on the current partial configuration and subsequent choices for factor-value assignments may subsequently force a may value to be inconsistent. Note also that may information only prunes values that are guaranteed to be inconsistent with the current assignment, thus preserving the full space of possible factor-value assignments for AETG and its t-set coverage heuristics to explore.

mAETG-History(CAModel)
Require: *uncovered-t-set-count*: calculated by initialization
1: $numCandidates = 50$
2: $numTests = 0$
3: $testCasePool = \emptyset$
4: **while** *uncovered-t-set-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=¬ factorInvolved(f) ∨ 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=¬ factorInvolved(f)∨ 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-t-set-count*)
17: increment $numTests$

Algorithm 3: AETG-History Algorithm

Must assignment information is calculated, at line 13a, and used, in lines 13b-13d, to enforce additional factor-value assignments that are common to all satisfiable

assignments that model extensions 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. This completely eliminates the need for AETG processing of those factors or for any subsequent SAT calls to confirm the satisfiability of those factor-value assignments. Our evaluation of the effectiveness of this optimization in [24] revealed that it reduced the number of SAT calls needed to compute a CMCA by an average of 59 percent across a set of 35 realistic CCIT problems.

4.3 Threshold Triggered SAT Assignments

Our evaluation of the AETG-History algorithm [24] indicated that it was very effective at reducing CCIT solution time with no negative effect on solution quality. Subsequent performance profiling revealed that less than 10 percent of AETG-History solution time was spent in formulating and solving SAT problems. Clearly, the bottleneck to further performance reduction lies in AETG processing.

The presence of constraints tends to reduce the size of the valid solution space and, as a row is built, this may lead to an increasingly limited set of valid choices of factor-values, especially late in the row. An expensive portion of an AETG-like algorithm is the method `selectBestValue`, line 12, which requires a linear scan of each possible value for the current factor and, for each value, it requires $\binom{j}{t}$ evaluations to compute how many new t-sets will be covered by that choice, where j is the current loop iteration starting at line 11. This requires a total of $v \times \binom{j}{t}$ computations for each call of this method. In the constrained portion of the search space that lies near the end of a row, the cost of this scan may yield little benefit since few consistent values may remain for a factor.

When the SAT solver finds a satisfying assignment, it calculates a complete configuration. That configuration may not, however, be one that drives the overall CCIT solution to a small CMCA—this is the intent of the AETG heuristics. The time needed to generate a CMCA can be reduced by short-circuiting the AETG calculations in lines 11–13 using the assignment calculated by the most recent successful SAT call. Fig. 4 illustrates this process when only one out of five, 20 percent, of the factors are assigned. The remaining four factor-value bindings are extracted from the satisfying assignment—illustrated by dotted arrows—and used to complete the row.

Short-circuiting AETG calculations early in a row can speed up solution times, but this may lead to larger CMCAs. Waiting until 100 percent of the factors are assigned yields no performance improvement, but no increase in CMCA size. For algorithmic frameworks like this, it is necessary to identify the parameter value that provides a desirable cost-benefit trade-off—we refer to this parameter as the row *threshold* and discuss finding a good value for the threshold in Section 5.

Algorithm 4 presents the AETG-Threshold algorithm. It differs from Algorithm 2 (AETG-SAT) only after the threshold has been reached. In the initialization step, the threshold value is input as a percentage of the row size and translated into the `threshold index`, the switching point in this algorithm. Then, in step 11, at 11aa, a check is made to determine if this threshold has been reached. If it has not, the algorithm continues as normal, allowing AETG to select the best symbol, followed by consistency checks using the SAT solver. Once the threshold value has been reached, execution switches and we mine the SAT assignment from the most recent SAT call (`mineRemainingAssignments`) and save this as the `satAssignedSet`. The entire assignment is then used to fill in the remaining factor-values without the use of the AETG strategy in lines 13b–13e. We note that the SAT solver makes random decisions at points in its search that are independent from that of the AETG-like algorithm.

mAETG-Threshold(CAModel)
Require: $uncovered\text{-}t\text{-}set\text{-}count$, $thresholdIndex$: 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=¬ factorInvolved(f) ∨ checkSAT($testCase_{count}$)
9: insertValueForFactor(l,f,$testCase_{count}$)
10: p=permuteRemainingFactors()
10a: $index$=1
11: **for** $f \in p$ **do**
11aa: **if** $index < thresholdIndex$
11a: sat=false
11b: $tries = 1, maxTries = v$
11c: **while** !sat and $tries \leq maxTries$
12: l=selectBestValue(f)
12a: sat=¬ factorInvolved(f)∨ checkSAT($testCase_{count}$)
12b: increment $tries$
13: insertValueForFactor(l,f,$testCase_{count}$)
13a: increment $index$
13b: **else**
13c: $satAssignedSet$=$mineRemainingAssignments$()
13d: **for**$(l, f) \in satAssignedSet$ **do**
13e: 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 4: AETG-Threshold Algorithm

4.4 Combining History and Threshold Optimizations

The history and threshold optimizations both seek to fill in multiple factor-value bindings in a single step. The advantage of the AETG-History is that it guaranteed not to interfere with AETG heuristics and, consequently, it will not increase CMCA size, as is possible with AETG-Threshold. On the other hand, since, in general, AETG-History only fills in a portion of the row, it will not reduce solution time as much as AETG-Threshold.

We consider a simple combination of these two algorithms which we call AETG-History-Threshold. The algorithm is not shown since it is a straightforward variation of Algorithm 3, which is used as the base algorithm up until a threshold has been reached for each row. After the threshold is reached, the algorithm switches strategy and mines the current SAT assignment to fill in the remaining factor-values as in Algorithm 4.

5 Empirical Investigation

We begin our empirical investigation by summarizing five case studies based on four software subjects that we first

conducted in [23], [24]. These show the abundance and types of constraints found in real software systems. We then present an analysis designed to evaluate the performance of the algorithms presented in Section 4 with respect to generation time and sample size. We utilize the five case studies and generate an additional set of synthesized CCIT problems for this analysis.

5.1 Case Studies

We have chosen four nontrivial highly-configurable software systems—SPIN [33], GCC [34], Apache [35], and Bugzilla [36] to study with respect to constraints. We analyzed the configuration options for these tools based on available documentation and constructed models of the options and any constraints among those options. All of our models should be considered an approximation of the true configuration space of the programs. One way we do this is by ignoring options we regard as overlapping, i.e., an option whose only purpose is to configure another set of options is ignored, as well as options that serve only to define program inputs. Another is by underestimating the number of possible values for each option. If an option takes an integer value in a certain range, we apply a kind of category partitioning and select a default value, a non-default legal value, and an illegal value; clearly, one could use more values to explore boundary values, but we choose not to do that. Similarly, for string options, we choose values modeling no string given, an empty string, and a legal string. Ultimately, the specific values chosen are determined during test input generation for a configuration, a topic we do not consider here. We report data on the size of these models, the number and variety of constraints, and the existence of implied forbidden t-sets.

5.1.1 SPIN Model Checker

SPIN is a widely used publicly available model checking tool [33]. SPIN serves both as a stable tool that people use to analyze the design of a system they are developing, expressed in SPIN's Promela language, and as a vehicle for research on advanced model checking techniques; as such, it has a large number and wide variety of options. We examined the manual pages for SPIN, available in [37], and used it as the primary means of determining options and constraints; in certain cases, we looked at the source code itself to confirm our understanding of constraints.

SPIN can be used in two different modes: as a *simulator* that animates a single run of the system description or as a *verifier* that exhaustively analyzes all possible runs of the described system. The "-a" options select verifier mode. The choice of mode also toggles between partitions of the remaining SPIN options, i.e., when simulator mode is selected, the verifier options are inactive and vice versa. While SPIN's simulator and verifier modes do share common code, we believe that the kind of bimodal behavior of SPIN warrants the development of two configuration models—one for each mode.

The simulator configuration model is the simpler of the two. It consists of 18 factors and, ignoring constraints, it could be modeled as an $MCA(N; 2, 2^{13}4^5)$, i.e., 13 binary options and five options each with four different values; this describes a space of 8.3×10^6 different system configurations. It has a total of 13 pairwise constraints that relate nine of the 18 factors. The nature of the interactions among the constraints for this problem, however, gives rise to no implied forbidden pairs. As for most problems, constraints for this problem can have a dramatic impact—enforcing just one of the 13 constraints eliminates over 2 million configurations.

The verifier configuration model is richer. It is worth noting that running a verification involves three steps: 1) A verifier implementation is *generated* by invoking the spin tool on a Promela input with selected command line parameters, 2) the verifier implementation is *compiled* by invoking a C compiler, for example, GCC, with a number of compilation flags, e.g., "-DSAFETY," to control the capabilities that are included in the verifier executable, and 3) finally, the verifier is *executed* with the option of passing several parameters. We view the separation of these phases as an implementation artifact and our verifier configuration model coalesces all of the options for these phases. This has the important consequence of allowing our model to properly account for constraints between configuration options in different phases. The model consists of 55 factors and, ignoring constraints, it could be modeled as a $MCA(N; 2, 2^{42}3^2 4^{11})$; this describes a space of 1.7×10^{20} different configurations. This model includes a total of 49 constraints—47 constraints that either require or forbid pairs of combinations of option values and two constraints over triples of such combinations. An example of a constraint is the illegality of compiling a verifier with the "-DSAFETY" flag and then executing the resulting verifier with the "-a" option to search for acceptance cycles; we note that these kinds of constraints are spread throughout software documentation and source code.

The set of SPIN verifier constraints span the majority of the factors in the model—33 of the 55 factors are involved in constraints. Furthermore, the interaction of these constraints through the model gives rise to nine implied forbidden pairs.

5.1.2 GCC Optimizer

GCC is a widely used compiler infrastructure that supports multiple input languages, e.g., C, C++, Fortran, Java, and Ada, and over 30 different target machine architectures. We analyzed version 4.1, the most recent release series of this large compiler infrastructure that has been under development for nearly 20 years. GCC is a very large system with over 100 developers contributing over the years and a steering committee consisting of 13 experts who strive to maintain its architectural integrity.

As was done for SPIN, we analyzed the documentation of GCC 4.1 [34] to determine the set of options and constraints among those options; in some cases, we ran the tool with different option settings to determine their compatibility. We selected a core component of GCC, the machine-independent *optimizer*, and modeled it with 199 factors and 40 constraints.

The optimizer model, without constraints, can be modeled as an $MCA(N; 2, 2^{189}3^{10})$; this describes a space of 4.6×10^{61} different configurations. Of the 40 constraints, three are three-way and the remaining 37 are pairwise. These constraints are related to 35 of the 199 factors and their interaction gives rise to two implied forbidden pairs.

Examples of constraints on optimizer settings include: "-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." We took the following approach to

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

	Factors and Values					Constraints					Factor Invol.	
	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 Cons Per Factor	2 Cons 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)

defining constraints. The commonly used "-O" options are interpreted as option packages that specify an initial set of option settings, but which can be overridden by an explicit "-fno" command. Interpreting these more strictly gives rise to hundreds of constraints, many of which are of higher order, i.e., they constrain three or more factor-values.

5.1.3 Apache HTTP Server 2.2

The Apache HTTP Server 2.2 is an open source widely used web server that works on both UNIX and Windows platforms. It can be customized by the system administrator through a set of *directives*. The directives for Apache fall into nine categories, which include the core program, extensions, server configuration, etc. In total, there are 379 configurable options that contribute to these categories. 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 six factors that were not part of the h directives. We added those options to our model for a total of 172 options. The final model has mostly binary options (92 percent) with a small number of factors that have between three and six options. The unconstrained Apache can be modeled as an $MCA(N; 2, 2^{158}3^{8}4^{4}5^{1}6^{1})$, i.e., there are 158 binary, eight ternary, four four-valued, and one factor each that have five and six values. This leads to an unconstrained configuration size of 1.8×10^{55}.

During our analysis, we uncovered seven constraints in the Apache documentation that relate between two and five different options. An example of a constraint for Apache is that the "Require" directive that selects which authenticated users can access a resource, must be accompanied by the "AuthName" and "AuthType" directives, as well as directives for "AuthUserFile" and "AuthGroupFile" (to define users and groups). Without these other directives being defined, "Requires" will not function properly. In total, only 18 options are involved in the seven constraints. Of these constraints, all but one are binary and one is a ternary. There are no implicit two-way constraints in this system.

5.1.4 Bugzilla 2.22.2

Bugzilla [36] is an open source defect tracking system from Mozilla. It provides developers with a mechanism to track outstanding bugs in their systems. The software includes advanced search capabilities, email notifications, multiple bug reporting formats, scheduled reports, time tracking, etc. It supports multiple database engines and is customizable by the user. After examining the documentation, we selected three sections of the user manual to which we have restricted our analysis. These are the sections that contain the core functionality: Chapter 3: *Administering Bugzilla*, Chapter 5: *Using Bugzilla*, and Chapter 6: *Customizing Bugzilla*. Our analysis uncovered 44 options.

When conducting our analysis, we found 10 additional options that were not included in one of the listed chapters, but that were somehow related through constraints to options within the scope of our analysis; we added these into our model to be complete. Our final model has 52 factors, of which 94.2 percent are binary. The final model for Bugzilla is an $MCA(N; 2, 2^{49}3^{1}4^{2})$. There are 49 binary, one ternary, and two four-valued factors. This leads to an unconstrained configuration space of 2.7×10^{16}. Bugzilla's documentation describes five constraints—four relating two options and one relating three options. An example of a Bugzilla constraint is, when the "Mail Transfer Agent" is set to "Postfix," it requires that the "sendmailnow" option be turned on. In total, 11 options were involved in the five constraints. We did not uncover any two-way implicit constraints for this system.

5.2 Synthesized CCIT Problems

The five case studies are essential elements of our evaluation, but they do not provide a large population of problems on which to compare algorithm performance. The time required to develop the case study models was significant and we felt that it was impractical to produce a significantly larger number of case studies in a timely fashion. Instead, we used the five case studies to develop a characterization of the abundance, type, and complexity of constraints found in real systems and then used that characterization to synthesize a large number of CCIT problems to include in our evaluation.

In Table 1, we provide a summary of the CCIT models for the five case studies, highlighting their main characteristics. The table shows counts of the number of factors (Num Factor) and explicit constraints (Num Cons) for each problem. It also provides the number and percentage (in parentheses) of factors with 2, 3, 4, or more values. Similarly, for constraints, it provides the number and percentage (in parentheses) of constraints of arity 2, 3, or more. As discussed in Section 4, it is possible to skip constraint processing during CMCA construction for factors that are not involved in constraints—the second column (Factor Invol.) under the Constraints subheading provides the number and percentage (in parentheses) of factors involved in constraints. The last two columns in the table show the dual of this information—they provide the number of constraints in which a factor participates. This provides an indication of the extent to which constraints are "coupled" and may give rise to implied constraints. For example, if a factor is involved in only a single constraint, it will fall into the first (1 Cons Per Factor) category. We do

not show data for factors involved in more than two constraints due to space limitations.

We use the summarization of case study characteristics to synthesize 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. The case studies had between 72 percent and 95 percent of their factors with only two values; 90 percent of the factors across all of the studies were binary. We skewed the number of binary factors toward the average across all case studies by selecting between 85 percent and 95 percent of the number of factors to be binary and the rest to involve between three and six factors. We weighted the latter decision with a 40 percent probability that three will be chosen and a 20 percent probability for the rest.

The ratio of constraints to factors in the case studies varied from 0.04 to 0.89, but this degree of variation leads to large numbers of models that bear no resemblance to the case studies. We chose to generate constraints by using the range of actual constraints, between 5 and 49, found in the case studies. Between 37 percent and 100 percent of the constraints are binary in our case studies; 90 percent of the constraints across all of the studies were binary. As with binary factors, we skewed the number of binary constraints toward the average across case studies by selecting 80-100 percent of constraints per problem to be binary. The remaining constraints were chosen as three, four, or five-way with equal probability. We used a greedy synthesis approach, so, at each decision point, if all constraints are assigned to a category, synthesis stops.

Another consideration that we tried to enforce is to make sure that between 40 percent and 100 percent of the factors involved in constraints are involved in only a single constraint while 10-20 percent of the factors are involved in two constraints; the latter range represents the skewing of factor involvement toward the average across all five case studies. Any constraints that are not bound to factors are configured to be involved with between three and nine constraints with equal probability.

We automated this approach to generate CCIT problems. In [24], we implemented a similar mechanism to generate data sets based on these case studies, but the approach described here refines that approach to better reflect the full range of possible factors, values, and constraints found in our case studies. Seventeen of the synthesized CCIT problems from [24] were judged to be consistent with the approach described here and we include them to allow comparability between the study results in this paper and [24].

We generated 24 new CCIT problems; five of those problems had inconsistent sets of constraints and were, thus, unsatisfiable. From the remaining 19 problems, we randomly selected 13 more for use in our evaluation. The CMCA models, numbers, and arity of constraints for all 30 synthesized CCIT problems are shown in Table 3. For each of the case studies and synthesized CCIT problems, it enumerates the factors for the CAModel and the constraints. This information is given in an abbreviated form that shows the numbers of factors with a given number of values in the form $\#values^{\#factors}$ and the number of constraints with a given arity in the form $arity^{\#constraints}$ (column No. Cons.).

5.3 Performance Evaluation

Our initial implementation of AETG-SAT [23] provided evidence indicating that our techniques perform as well as or better than existing constraint handling techniques. The overhead required to find solutions using this technique was, however, nontrivial. In further work [24], we explored the use of incremental SAT solving and developed the AETG-History algorithm. Evaluation of these enhancements indicated that they could significantly reduce CMCA generation time with no negative impact on the quality of solutions.

In this section, we compare the performance of AETG-History, AETG-Threshold, and AETG-History-Threshold using an incremental SAT version of AETG-SAT as the baseline. Our goal is to empirically evaluate the algorithms with respect to both the computational time required (efficiency) and the size (quality) of the resulting solutions. Before comparing AETG-History and AETG-Threshold, however, it is first necessary to select a *threshold value*. Our first study evaluates various threshold values to find the best balance of efficiency and quality in our data samples.

5.4 Finding a Good Threshold Point

The AETG-Threshold algorithm triggers a switch in the algorithmic behavior at a given threshold point. Once this threshold has been reached, the algorithm stops any further AETG evaluations and instead fills in all of the remaining factor-values using the last satisfying assignment found by the SAT solver. We expect a range of behavior for this algorithm. When the threshold is set very low, we expect solutions that are effectively random in achieving interaction coverage, while thresholds closer to the end will likely save little computation. Since the selection of the threshold will affect our results, we compare both time and size over a range of threshold values. To determine a threshold, we randomly selected five samples from our synthesized data. These are sample numbers 14, 15, 21, 28, and 29 from Table 3. We evaluate threshold in 10 percent increments from 10 percent to 90 percent. For each threshold, we run AETG-Threshold 50 times and collect both the time in seconds as well as the size of the resulting CMCA. Initialization is performed once and this time is divided evenly among samples. Fig. 5 shows box plots for `time` and `size` for each of the 50 runs summed across these five samples. The graph on the left plots size, while the graph on the right plots execution time. These plots capture the variability in the 50 runs of the algorithm, while showing the median total times and sizes based on different threshold values.

As expected, the CMCA sizes for low thresholds are large compared with the sizes calculated for a threshold of 50 percent or greater. The runtimes are dramatically lower for thresholds below 50 percent and runtime increases rapidly with increasing threshold.

Visual inspection of the plots suggests that a threshold in the 50-70 percent range provides a good balance between speed and CMCA size. We are interested in confirming this intuition by calculating the threshold that provides the best cost-benefit trade-off. We use a normalization technique to equalize the values for time and size to contribute equally in our decision. It is possible to associate more weight to one or the other of these metrics, depending on the final objective, but, for our initial investigation, we chose an

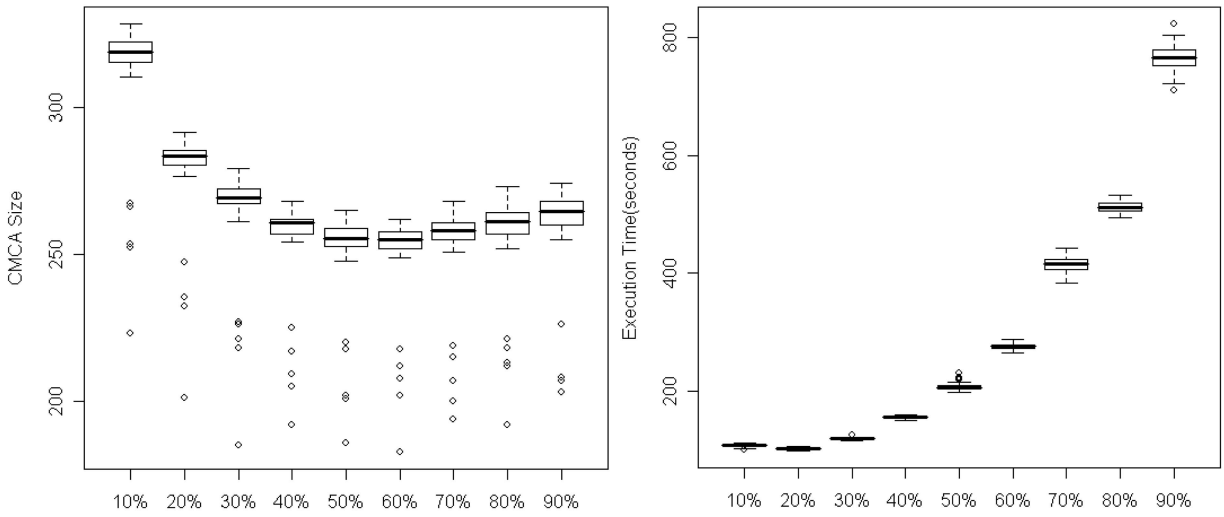

Fig. 5. SAT threshold performance for five random samples.

TABLE 2
Time and Size of Five Samples for Threshold Percentages

	Threshold Percentage								
	10%	**20%**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**
Time	109.33	103.04	120.20	155.59	207.94	276.61	415.37	512.97	765.69
$timeRatio$	0.01	0.00	0.03	0.08	0.16	0.26	0.47	0.62	1.00
Size	312.64	278.64	264.56	255.60	251.22	250.42	253.62	256.94	259.96
$sizeRatio$	1.00	0.45	0.23	0.08	0.01	0.00	0.05	0.10	0.15
$combinedTimeSize$	10.66	4.83	2.45	0.97	0.30	**0.26**	1.02	1.73	2.63

equal contribution. Given the different scales, we reduce the impact of each to a relative importance. We first calculate the *timeRatio* and *sizeRatio* by subtracting the minimum time (or size) from the time (or size) of each threshold point. For instance, in the $timeRatio$, all times will subtract 103.0, as is seen in Table 2. We then divide this number by the range of times (or range of sizes). This gives us a number between zero and one for each ratio. Zero means that the time (or size) matches the minimum value for all thresholds and one means it matches the maximum time (or size). We use a weighted sum of these, $combinedTimeSize$, and select the minimum value to set our best threshold. Since the ranges of data for time and size vary by a factor of $sizeRange/timeRange$ (10.7 in our data set given a time range of 662.7 and a size range of 62.2), we multiply the size ratio by this value, giving us the following formula:

$$combinedTimeSize = timeRatio + (sizeRange/timeRange) \times sizeRatio.$$

The data for these calculations for the five samples is given in Table 2. We use the average of the sum of the 50 data repeats as a basis for this calculation. The results of this analysis show that the threshold value of 60 percent provides the best balance of both time and size. In Table 3, we present only the data for this threshold value.

5.5 Comparing Algorithms

We compare the four variations of AETG-SAT presented in Section 4, with a threshold of 60 percent for AETG-Threshold and AETG-History-Threshold and the unconstrained AETG algorithm. All of our implementations are written in C++ and use miniSAT v1.14.1 written in C [32]. All program runtime data is gathered by executing implementations on an Opteron 250 processor with 4 Gbytes of RAM running the Fedora Core 3 installation of Linux.

For each technique and CCIT problem, we ran 50 trials; this helps account for the random variation that is inherent in AETG-like algorithms. We collect both CMCA size and execution times for each of the 50 trials. Once again, we divide the initialization times evenly among all runs. Table 3 shows the results of generating samples for $t = 2$ for the five case studies as well as the 30 synthesized CCIT problems. The first three columns identify the CCIT problem and characterize the CAModel and problem constraints. The remaining columns are split into two groups of five, reporting for each of the five techniques in terms of CMCA size and execution time in seconds; each cell in the table gives the average over the 50 trials for either size or time. The last row of the table is the sum of averages across all data sets.

The variation in covering array size across techniques is relatively modest. It is noteworthy that less than 3 percent of the MCA rows produced by the unconstrained AETG technique satisfy constraints. AETG-SAT and AETG-History produce nearly identical sizes as do AETG-Threshold and AETG-History-Threshold. A difference of three rows between AETG-Threshold and AETG-History-Threshold across the more than 1,500 is within the expected variation attributable to randomization in AETG, as can be observed by the range in the box plots of the CMCA size data in Fig. 5. The 60 percent Threshold algorithms appear to provide a modest reduction, approximately 3 percent in CMCA size. We conjecture that the effectively random

TABLE 3
Average Size and Time over 50 Runs

Covering Array, t=2			Size					Time in Secs				
	CAModel	No. Cons.	mAETG size/ SAT rows	AETG SAT	AETG Hist	AETG Thres 60%	AETG H-T 60%	mAETG	AETG SAT	AETG Hist	AETG Thres 60%	AETG H-T 60%
SPIN-S	$2^{1}3^{4}5^{}$	2^{13}	26.3/8	27.1	27.1	**27.0**	27.1	0.3	0.4	0.3	0.2	**0.2**
SPIN-V	$2^{42}3^{2}4^{11}$	$2^{47}3^{2}$	36.0/0	**42.5**	42.7	45.0	44.9	8.2	11.3	8.5	4.5	**3.5**
GCC	$2^{189}3^{10}$	$2^{37}3^{3}$	25.2/1	24.8	**24.7**	26.3	26.3	221.7	286.9	204	70.2	**68.0**
Apache	$2^{158}3^{8}4^{4}5^{1}6^{1}$	$2^{3}3^{1}4^{2}5^{1}$	42.4/14	42.8	42.5	42.6	42.9	258.3	249.2	244.1	**76.4**	76.5
Bugz.	$2^{49}3^{1}4^{2}$	$2^{4}3^{1}$	25.0/7	24.9	25.2	**21.8**	22.2	4.5	6.2	4.4	1.9	**1.5**
1.	$2^{86}3^{3}4^{1}5^{5}6^{2}$	$2^{20}3^{3}4^{1}$	56.5/0	54.7	55.4	**53.3**	53.7	79.1	71.3	66.1	24.4	**22.2**
2.	$2^{86}3^{3}4^{3}5^{1}6^{1}$	$2^{19}3^{3}$	40.0/0	40.1	**39.7**	40.4	40.5	39.9	47	40.7	16.2	**14.7**
3.	$2^{27}4^{2}$	$2^{9}3^{1}$	23.1/0	21.0	21.2	**20.7**	20.8	0.9	0.9	0.8	0.4	**0.3**
4.	$2^{51}3^{4}4^{2}5^{1}$	$2^{15}3^{2}$	29.8/1	28.7	28.7	28.9	**28.6**	8.8	8.8	8.1	3.4	**3.1**
5.	$2^{155}3^{7}4^{3}5^{5}6^{4}$	$2^{32}3^{6}4^{1}$	65.8/0	64.1	64.5	**63.8**	64.3	404.3	404.8	372.9	134.8	**120.3**
6.	$2^{73}4^{3}6^{1}$	$2^{26}3^{4}$	34.3/0	34.0	**33.9**	34.0	34.0	21.0	25.1	22.1	8.2	**7.2**
7.	$2^{29}3^{1}$	$2^{13}3^{2}$	12.4/0	12.0	**12.1**	12.5	12.5	0.6	0.6	0.6	**0.3**	**0.3**
8.	$2^{109}3^{2}4^{2}5^{3}6^{3}$	$2^{32}3^{4}4^{1}$	59.4/0	57.3	57.1	**55.6**	56.1	117	131.8	148.3	45.1	**40.6**
9.	$2^{57}3^{1}4^{1}5^{1}6^{1}$	$2^{30}3^{7}$	36.3/0	27.2	27.3	26.1	**26.0**	10.6	9.4	8.2	3.5	**3.0**
10.	$2^{130}3^{6}4^{5}5^{2}6^{4}$	$2^{40}3^{7}$	64.0/0	64.1	63.9	**60.3**	60.4	257	261.5	230.5	82.8	**74.3**
11.	$2^{84}3^{4}4^{2}5^{2}6^{4}$	$2^{28}3^{4}$	61.5/0	61.1	60.6	59.0	**58.3**	67.7	80.2	68.9	26.8	**23.8**
12.	$2^{136}3^{4}4^{3}5^{1}6^{3}$	$2^{23}3^{4}$	58.8/0	57.1	56.8	**54.3**	54.5	235.0	228.3	212.8	75.6	**68.0**
13.	$2^{124}3^{4}4^{1}5^{2}6^{2}$	$2^{22}3^{4}$	51.5/0	51.0	51.8	49.3	**48.6**	145.2	153.2	146.8	60.1	**45.5**
14.	$2^{81}3^{5}4^{3}6^{3}$	$2^{13}3^{2}$	56.6/3	56.2	56.0	**51.7**	51.8	54.3	61.4	57.7	20.6	**20.2**
15.	$2^{50}3^{4}4^{1}5^{2}6^{1}$	$2^{20}3^{2}$	41.1/1	40.7	40.7	**40.4**	40.7	12.3	13.0	11.7	4.9	**4.7**
16.	$2^{81}3^{3}4^{2}6^{1}$	$2^{30}3^{4}$	33.4/0	**33.0**	33.1	33.1	33.4	26.9	35.1	27.7	11.1	**9.7**
17.	$2^{128}3^{3}4^{2}5^{1}6^{3}$	$2^{25}3^{4}$	57.1/0	57.0	56.6	53.6	**53.4**	169.6	173.1	156.3	52.0	**46.9**
18.	$2^{127}3^{2}4^{4}5^{6}6^{2}$	$2^{23}3^{4}4^{1}$	59.5/0	57.7	58.3	**57.2**	57.3	198.3	199.2	193.1	67.3	**60.1**
19.	$2^{172}3^{9}4^{9}5^{3}6^{4}$	$2^{38}3^{5}$	68.0/0	66.3	65.7	**64.4**	64.7	610.7	586	534.4	206.6	**168.4**
20.	$2^{138}3^{4}4^{5}5^{4}6^{7}$	$2^{42}3^{6}$	72.6/0	71.6	71.8	**71.5**	71.8	347.9	362.9	325.7	121.1	**109.9**
21.	$2^{76}3^{3}4^{2}5^{1}6^{3}$	$2^{40}3^{6}$	56.0/0	55.0	54.7	**51.3**	51.7	42.1	49.6	41.6	18.5	**14.3**
22.	$2^{73}3^{3}4^{3}$	$2^{31}3^{4}$	28.8/0	28.7	28.8	**26.2**	26.4	17.3	19.9	17.6	6.7	**5.9**
23.	$2^{25}3^{1}6^{1}$	$2^{13}3^{2}$	20.7/1	**15.7**	**15.7**	16.3	16.0	0.8	0.7	0.6	**0.3**	**0.3**
24.	$2^{110}3^{2}5^{3}6^{4}$	$2^{25}3^{4}$	61.0/0	59.9	60.0	**58.1**	58.3	124.2	134.7	128.9	46.4	**40.6**
25.	$2^{118}3^{6}4^{2}5^{2}6^{6}$	$2^{23}3^{3}4^{1}$	67.6/0	66.2	66.4	**65.7**	**65.7**	193.6	196.7	180.5	**63.4**	61.7
26.	$2^{87}3^{1}4^{3}5^{4}$	$2^{28}3^{4}$	44.2/0	43.3	43.5	**42.0**	**42.0**	45.5	50.6	47.5	20.7	**15.5**
27.	$2^{55}3^{2}4^{2}5^{1}6^{2}$	$2^{17}3^{3}$	49.8/0	49.8	50.5	46.6	**45.8**	15.7	20.0	15.1	5.4	**4.6**
28.	$2^{167}3^{16}4^{2}5^{3}6^{6}$	$2^{31}3^{6}$	70.1/0	68.4	68.7	**68.4**	68.5	584.5	588.5	546.3	180.9	**170.4**
29.	$2^{134}3^{7}5^{3}$	$2^{19}3^{3}$	42.2/0	41.2	41.4	38.6	**38.4**	148.1	142.7	135.9	51.7	**38.9**
30.	$2^{72}3^{4}4^{1}6^{2}$	$2^{20}3^{2}$	49.2/0	50.2	50.4	**45.6**	45.8	29.8	35.8	31.1	11.8	**10.3**
sum			1626.4/36	1595.4	1597.7	**1551.6**	1553.3	4501.3	4646.6	4249.2	1524.3	**1356.3**

selection made after the threshold point provides a relaxation in the aggressive one-row-at-a-time greedy technique, allowing better decisions to be made in later rows. We know that metaheuristic search techniques that construct the entire array at a time (rather than fix one-row-at-time) and relax intermediate solutions by allowing occasional "bad choices" in general produce smaller covering arrays for both unconstrained and CCIT problems [9], [23]. Further analysis is needed to confirm this conjecture.

The variation in execution time data across techniques is more significant. AETG-History yields a 9 percent reduction in solution time over AETG-SAT and a 5 percent reduction in solution time over unconstrained AETG; these results are consistent in [24]. AETG-Threshold, as expected, significantly speeds up solution time by skipping AETG processing on 40 percent of each row; it yields a 67 percent reduction in solution time relative to AETG-SAT.

The AETG-History-Threshold shows that the History and Threshold optimizations target different aspects of AETG-SAT algorithm since the data reveal that their benefits accumulate when the optimizations are composed. The AETG-History-Threshold technique yields a 71 percent reduction in solution time relative to AETG-SAT. We believe that additional improvements are possible by more tightly integrating History and Threshold. The current combination attempts to clearly separate the portions of the row in which each technique is active; however, it is possible to accelerate progress toward the threshold by counting the must assignments produced by History as making progress through the row. This integration would further reduce solution time without impacting solution quality—since Threshold will not overwrite must assignments from History.

These data demonstrate unequivocally that integrating constraints into CCIT solution algorithms can not only be efficient, but can actually make solution times significantly faster. This result may seem counterintuitive at first, but it can be attributed to the fact that the techniques leverage SAT to aggressively prune the AETG search space. The cost benefits of this pruning more than compensate for the additional overhead of mining SAT solver data structures, maintaining threshold counters, and identifying implicit constraints during initialization.

5.6 Further Analysis of the Threshold

We performed additional analyses to examine the impact of increasing the covering array strength and of changing the characterization of the covering array on the threshold. First, we examine the threshold when our algorithms are run for higher strength, i.e., different values of t. When we run all of the same samples for $t = 3$, we see the threshold move to 80 percent (this threshold analysis is not shown).

TABLE 4
Average Size and Time over 50 Runs for $t = 3$

Covering Array, t=3			Size					Time in Secs				
	CAModel	No. Cons.	mAETG	AETG SAT	AETG Hist	AETG Thres 80%	AETG H-T 80%	mAETG	AETG SAT	AETG Hist	AETG Thres 80%	AETG H-T 80%
SPIN-S	$2^{13}4^5$	2^{13}	105.9	**117.5**	117.1	118.1	118.4	5.9	7.7	6.2	4.9	**4.1**
SPIN-V	$2^{42}3^24^{11}$	$2^{47}3^2$	168.3	243.0	**242.2**	254.1	253.7	611.2	879.4	636.9	490.4	**394.8**
GCC	$2^{189}3^{10}$	$2^{37}3^3$	88.5	**106.5**	106.9	111.8	112.3	44498.2	54307.2	53226.8	28082.1	**27102.1**
Apache	$2^{158}3^84^45^16^1$	$2^33^14^25^1$	207.4	**206.9**	206.8	212.3	211.7	60126.4	60216.3	60785.2	31210.1	**30066.0**
Bugz.	$2^{49}3^14^2$	2^43^1	71.1	**71.7**	72.2	74.8	75.3	199.4	209.6	190.1	123.9	**103.8**
sum			641.3	745.7	**745.2**	771.2	771.4	105441.1	115620.2	114845.2	59911.3	**57670.7**

We also see a steady decrease in the resulting covering array size. This differs from the $t = 2$ data, where our size was minimal at the 60 percent threshold. Our conjecture is that the random choices made by the SAT solver affect a much larger set when $t = 3$, which means we are less likely to make good choices by chance. Table 4 shows time and size data for our case study subjects when $t = 3$. Although our threshold is now 80 percent, we still see a large time savings when we use AETG-Threshold. This is because the runtimes are much longer overall. For instance in GCC, the time saved by using an 80 percent threshold is approximately 8 hours. Our smallest arrays are found with AETG-History for $t = 3$. Which algorithmic technique provides the greatest cost savings will now depend on the cost of running tests for a single configuration versus the cost of constructing the covering array.

We also examined synthesized data sets for $t = 2$ that do not follow the characteristics of our case study subjects. Specifically, we decreased the number of factors, but increased the average number of values for each factor to be between 3 and 30 values. In this situation, we also saw an increase in the threshold to 80 percent, leading us to conclude that the threshold will be sensitive to the parameters of the constrained covering array. Consequently, we believe that selection of specific threshold values is best determined by balancing the value of CMCA size versus generation time in a particular testing context.

5.7 Threats to Validity

The most significant threats to our findings is their generalization to other subjects. We cannot be sure that the subjects chosen and the respective simulated data are representative of all configurable software. We have, however, tried to control for this by using four different subjects from differing domains that have large user population bases. All of these are open source programs, however, which may not be reflective of proprietary systems. We have also seen that the size data of our resulting samples and the resulting threshold is sensitive to the parameters of the covering array. This means that both the software being modeled and the strength of the array may affect the threshold value. The time data, however, appears to be more stable across all experiments and should generalize better.

We have taken special measures to assure the internal validity of our findings. We have independently checked the constrained covering arrays through random sampling with different programs to confirm that they generate the correct constrained covering arrays. We have also validated the programs that are used to perform that checking.

While it is clear that one could develop many measures for judging the value of a CCIT method, we believe that CMCA size and generation time are the core elements of such an evaluation. Their relative weights in drawing value judgments may be varied depending on usage context, but we leave such exploration to future work.

6 Related Work

Although few algorithmic solutions for constraints in CIT have been presented in the literature, their existence has been discussed by others. Hartman formalizes CCIT where he defines forbidden configurations [28]. Our use of forbidden tuples can be thought of as partial forbidden configurations. Bryce and Colbourn [22] differentiate between forbidden combinations and combinations that should be *avoided* but may legally be present. They term these *soft constraints*.

Implicit constraints have been discussed and handled in a few different ways in the research community. The difficulties are discussed by the designers of AETG [6], [12], although they do not suggest an algorithmic extension to AETG to incorporate these during greedy construction. Rather, they remodel the input and/or postprocess their resulting CMCA samples. PICT is a Microsoft internal tool [10]. It uses an AETG-like algorithm optimized for speed. Specifically, it sets $numCandidates = 1$ in line 1 of Algorithm 1 (Section 4). This may reduce the quality of the resulting solution since the default of generating 50 candidates for each configuration in AETG provides a *set* of choices each time from which the best can be selected. In [10], a discussion of the algorithm and constraint handling states that, as they build their solutions, they *check* to see if the solution is legal, but the paper does not provide detail about how this check is actually implemented or how implicit constraints are uncovered.

Some tools expose implicit constraints by requiring the user to expand or partition the input. One tool that requires an expansion is the Intelligent Test Case Generator (Whitch) [21] that includes two algorithms for finding covering arrays, TOFU and Combinatorial Test Services (CTS) [28]. The TestCover service [20] uses *direct product block notation* to identify a set of allowed test cases computed as a direct product of compatible factor-values. It takes a collection of these products as input data to define the set of all allowed test cases, implicitly defining the constraints. The construction techniques used in TestCover are primarily mathematical constructions and are therefore specific to covering arrays with particular relationships between different parameter combinations of t, k, and $|v|$.

In [22], the authors extend the AETG-like deterministic density algorithm (DDA) [13] to include constraint handling by weighting tuples as desirable or not. They use this method to *avoid* tuples *if possible*. When implicit constraints exist, their algorithms will fail to avoid them. Although they provide a discussion of *hard constraints* which match our forbidden tuples, their algorithm does not attempt to handle these. Additionally, their work is limited to the avoidance of forbidden pairs.

Hnich et al. [15] use constraint solving to *build* unconstrained CIT samples by translating the definition into a Boolean satisfiability problem. They encode at-least and at-most constraints as disjunctive clauses, as well as each of the required t-sets that must be covered. Their approach has limited scalability to realistically sized problems due to the exponentially large constraint set. Although they use a SAT solver to directly construct a CIT sample, they refer to the forbidden tuples discussed in this paper as "side-constraints" and leave the handling of these as a future extension. Our work uses the SAT solver only for the base constraints, while it uses the AETG algorithm to handle the required t-sets.

A significant emphasis of the computer-aided verification community recently has been the integration of various decision procedures into the SAT solving framework. These Satisfiability Modulo Theory (SMT) approaches (e.g., [38]) use SAT as a backbone and integrate other decision procedures as reasoning steps. Our work differs from this approach because we use AETG as the driving application in interacting with SAT. Furthermore, we extract intermediate results from SAT computations for AETG, whereas, in SMT, the decision procedures are run to completion and then only their results are consumed. Thus, our work represents a finer and domain-specific integration of SAT techniques as opposed to the coarser and general SMT approach.

Finally, we place our work in a larger context of a body of work that solves optimization problems arising in software engineering, commonly termed *search-based software engineering*. There has been a lot of recent research activity that leverages both greedy and metaheuristic search techniques for tackling problems that cannot be solved through exhaustive enumeration. We do not present a complete survey of that literature here, but refer readers to [39] for a more thorough discussion. Both greedy and metaheuristic techniques use heuristics to guide the search toward a "good," although perhaps less than optimal, solution. The two classes of algorithms differ in that the greedy techniques are often less expensive computationally and may be easier to implement while often providing solutions that are not as close to optimal as their metaheuristic counterparts. The choice of algorithm will depend on trade-offs that weigh the cost of the algorithm, the complexity of the search terrain, and requirements of the problem solution.

Metaheuristic search formulates the problem as an objective function subject to a set of conditions. Common metaheuristic search algorithms include genetic algorithms, simulated annealing, and tabu search. Examples of search activity in software testing include the use of genetic algorithms for test case generation [40], [41], [42], greedy algorithms for regression test case selection and prioritization [43], and our own work that uses simulated annealing for generation of CIT samples [9], [44]. Recent work on multiobjective optimization, such as that in [45], has commonality with the work in this paper because it provides a method to handle multiple orthogonal search objectives at once. The primary goal of most of the search-based research, however, has been to solve one particular optimization problem and has not addressed ways to incorporate additional external problem constraints such as the type that we discuss in this paper. We believe there is significant promise in integrating SAT tightly with other search-based CIT methods. Metaheuristic search methods such as simulated annealing offer the potential for generating smaller CIT samples than greedy methods [9] and a basic integration of SAT with search appears to offer significant opportunities for optimization [23].

7 Conclusions and Future Work

The conventional wisdom in the CIT community is that constraints significantly complicate the problem of computing a CIT sample. In this paper, we have presented a set of algorithms that synergistically integrate Boolean satisfiability algorithms with greedy CIT generation algorithms. The most efficient of these algorithms allows high-quality CIT samples to be computed in less than one-third the time of widely used unconstrained CIT algorithms. Moreover, this performance benefit was observed on a collection of CCIT problems that reflect the richness of constraints found in real-world systems. We believe this represents a promising step in advancing CIT methods toward even broader applicability for the testing of highly-configurable software systems.

The key insight in this work is to leverage the fact that both CIT generation and SAT solver algorithms perform a search of the same space. By formulating CCIT sample generation as alternating phases of CIT and SAT search, we leverage information from one search to inform the other. This leads to significant pruning of the CIT search and reductions in execution time, while retaining the portions of the search space that contain high-quality solutions.

We believe that the techniques in this paper open the way for more aggressive scaling of the application of CCIT methods. In addition to scaling the size of subjects, an additional, and orthogonal, dimension of scaling is to consider higher CIT strength. While our evaluation considered mostly pairwise CCIT, it is likely that, for mission-critical systems, engineers will target higher order coverage, which will dramatically increase the cost of CIT. The analyses we performed on some instances of CCIT for $t = 3$ in [23] and in this paper indicate that runtimes are dramatically larger than for $t = 2$. We also observed that the threshold point for retaining high-quality solutions increases. We have not yet explored the impact on strengths of $t > 3$. More study is needed to better understand the scalability of our CCIT methods to extremely large-scale highly-configurable mission-critical systems.

Acknowledgments

The authors thank the following for helpful comments on this subject: Alan Hartman, Tim Klinger, Christopher Lott, and George Sherwood. This work was supported in part by an EPSCoR FIRST award and by the US Army Research Office through DURIP award W91NF-04-1-0104 and by the US National Science Foundation through Awards 0747009, 0720654, 0541263, 0429149, and 0454203. Any opinions,

findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US Army Research Office and US National Science Foundation.

REFERENCES

[1] D.L. Parnas, "On the Design and Development of Program Families," *IEEE Trans. Software Eng.*, vol. 2, no. 1, pp. 1-9, 1976.

[2] D. Dvorak, R. Rasmussen, G. Reeves, and A. Sacks, "Software Architecture Themes in JPL's Mission Data System," *Proc. IEEE Aerospace Conf.*, Mar. 2000.

[3] D. Kuhn, D.R. Wallace, and A.M. Gallo, "Software Fault Interactions and Implications for Software Testing," *IEEE Trans. Software Eng.*, vol. 30, no. 6, pp. 418-421, June 2004.

[4] C. Yilmaz, M.B. Cohen, and A. Porter, "Covering Arrays for Efficient Fault Characterization in Complex Configuration Spaces," *IEEE Trans. Software Eng.*, vol. 31, no. 1, pp. 20-34, Jan. 2006.

[5] Nokia Corporation, "Nokia Mobile Phone Line," http://www.nokiausa.com/phones, 2007.

[6] 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 Trans. Software Eng.*, vol. 23, no. 7, pp. 437-444, July 1997.

[7] R. Brownlie, J. Prowse, and M.S. Phadke, "Robust Testing of AT&T PMX/StarMAIL Using OATS," *AT&T Technical J.*, vol. 71, no. 3, pp. 41-47, 1992.

[8] R.C. Bryce, C.J. Colbourn, and M.B. Cohen, "A Framework of Greedy Methods for Constructing Interaction Test Suites," *Proc. 27th Int'l Conf. Software Eng.*, pp. 146-155, May 2005.

[9] M.B. Cohen, C.J. Colbourn, P.B. Gibbons, and W.B. Mugridge, "Constructing Test Suites for Interaction Testing," *Proc. 25th Int'l Conf. Software Eng.*, pp. 38-48, May 2003.

[10] J. Czerwonka, "Pairwise Testing in Real World," *Proc. 24th Pacific Northwest Software Quality Conf.*, pp. 419-430, Oct. 2006.

[11] I.S. Dunietz, W.K. Ehrlich, B.D. Szablak, C.L. Mallows, and A. Iannino, "Applying Design of Experiments to Software Testing," *Proc. 19th Int'l Conf. Software Eng.*, pp. 205-215, 1997.

[12] C. Lott, A. Jain, and S. Dalal, "Modeling Requirements for Combinatorial Software Testing," *Proc. First Int'l Workshop Advances in Model-Based Testing*, pp. 1-7, May 2005.

[13] C.J. Colbourn, M.B. Cohen, and R.C. Turban, "A Deterministic Density Algorithm for Pairwise Interaction Coverage," *Proc. IASTED Int'l Conf. Software Eng.*, pp. 345-352, Feb. 2004.

[14] A. Hartman and L. Raskin, "Problems and Algorithms for Covering Arrays," *Discrete Math.*, vol. 284, pp. 149-156, 2004.

[15] B. Hnich, S. Prestwich, E. Selensky, and B. Smith, "Constraint Models for the Covering Test Problem," *Constraints*, vol. 11, pp. 199-219, 2006.

[16] K. Nurmela, "Upper Bounds for Covering Arrays by Tabu Search," *Discrete Applied Math.*, vol. 138, nos. 1-2, pp. 143-152, 2004.

[17] K.C. Tai and Y. Lei, "A Test Generation Strategy for Pairwise Testing," *IEEE Trans. Software Eng.*, vol. 28, no. 1, pp. 109-111, Jan. 2002.

[18] Y. Tung and W.S. Aldiwan, "Automating Test Case Generation for the New Generation Mission Software System," *Proc. IEEE Aerospace Conf.*, pp. 431-437, 2000.

[19] A.W. Williams and R.L. Probert, "A Measure for Component Interaction Test Coverage," *Proc. ACS/IEEE Int'l Conf. Computer Systems and Applications*, pp. 301-311, Oct. 2001.

[20] G. Sherwood, "Testcover.com," http://testcover.com/pub/constex.php, 2006.

[21] IBM alphaWorks, "IBM Intelligent Test Case Handler," http://www.alphaworks.ibm.com/tech/whitch, 2005.

[22] R.C. Bryce and C.J. Colbourn, "Prioritized Interaction Testing for Pair-Wise Coverage with Seeding and Constraints," *J. Information and Software Technology*, vol. 48, no. 10, pp. 960-970, 2006.

[23] M.B. Cohen, M.B. Dwyer, and J. Shi, "Interaction Testing of Highly-Configurable Systems in the Presence of Constraints," *Proc. Int'l Symp. Software Testing and Analysis*, pp. 129-139, July 2007.

[24] M.B. Cohen, M.B. Dwyer, and J. Shi, "Exploiting Constraint Solving History to Construct Interaction Test Suites," *Proc. Testing: Academic and Industrial Conf.—Practice and Research Techniques*, pp. 121-130, Sept. 2007.

[25] M.B. Cohen, M.B. Dwyer, and J. Shi, "Coverage and Adequacy in Software Product Line Testing," *Proc. ISSTA Workshop Role of Software Architecture for Testing and Analysis*, pp. 53-63, July 2006.

[26] R. Mandl, "Orthogonal Latin Squares: An Application of Experiment Design to Compiler Testing," *Comm. ACM*, vol. 28, no. 10, pp. 1054-1058, Oct. 1985.

[27] J.D. McGregor, "Testing a Software Product Line," technical report, Software Eng. Inst., Carnegie Mellon Univ., Dec. 2001.

[28] A. Hartman, "Software and Hardware Testing Using Combinatorial Covering Suites," *Proc. Graph Theory, Combinatorics and Algorithms: Interdisciplinary Applications*, pp. 266-327, 2005.

[29] M. Davis and H. Putnam, "A Computing Procedure for Quantification Theory," *J. ACM*, vol. 7, no. 3, pp. 201-215, 1960.

[30] J.P. Marques-Silva and K.A. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability," *IEEE Trans. Computers*, vol. 48, no. 5, pp. 506-521, May 1999.

[31] T. Walsh, "SAT v CSP," *Proc. Int'l Conf. Principles and Practice of Constraint Programming*, vol. 1894, pp. 441-456, Sept. 2000.

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

[33] G.J. Holzmann, "The Model Checker SPIN," *IEEE Trans. Software Eng.*, vol. 23, no. 5, pp. 279-295, May 1997.

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

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

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

[37] G.J. Holzmann, "On-the-Fly, LTL Model Checking with SPIN: Man Pages," http://spinroot.com/spin/Man/index.html, 2006.

[38] C. Tinelli, "A DPLL-Based Calculus for Ground Satisfiability Modulo Theories," *Proc. Eighth European Conf. Logics in Artificial Intelligence*, G. Ianni and S. Flesca, eds., pp. 308-319, 2002.

[39] M. Harman, "The Current State and Future of Search Based Software Engineering," *Proc. Future of Software Eng.*, pp. 342-357, 2007.

[40] B. Korel, "Automated Software Test Data Generation," *IEEE Trans. Software Eng.*, vol. 16, no. 8, pp. 870-879, Aug. 1990.

[41] P. McMinn, "Search-Based Software Test Data Generation: A Survey," *J. Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105-156, 2004.

[42] R. Pargas, M.J. Harrold, and R. Peck, "Test-Data Generation Using Genetic Algorithms," *J. Software Testing, Verification and Reliability*, vol. 9, no. 3, pp. 263-282, 1999.

[43] Z. Li, M. Harman, and R.M. Hierons, "Search Algorithms for Regression Test Case Prioritization," *IEEE Trans. Software Eng.*, vol. 33, no. 4, pp. 225-237, Apr. 2007.

[44] M.B. Cohen, C.J. Colbourn, and A.C.H. Ling, "Augmenting Simulated Annealing to Build Interaction Test Suites," *Proc. 14th IEEE Int'l Symp. Software Reliability Eng.*, pp. 394-405, Nov. 2003.

[45] S. Yoo and M. Harman, "Pareto Efficient Multi-Objective Test Case Selection," *Proc. Int'l Symp. Software Testing and Analysis*, pp. 140-150, July 2007.

Myra B. Cohen received the BS degree from the School of Agriculture and Life Sciences, Cornell University, the MS degree in computer science from the University of Vermont, and the PhD degree in computer science from the University of Auckland, Auckland, New Zealand. She is an assistant professor in the Department of Computer Science and Engineering at the University of Nebraska-Lincoln, where she is a member of the Laboratory for Empirically based Software Quality Research and Development (ESQuaReD). She is a recipient of a US National Science Foundation Faculty Early CAREER Development Award. Her research interests include testing of configurable software systems, combinatorial interaction testing, and search-based software engineering. She is a member of the IEEE and the ACM.

Matthew B. Dwyer received the BS degree in electrical engineering from the University of Rochester in 1985, the MS degree in computer science from the University of Massachusetts at Boston in 1990, and the PhD degree from the University of Massachusetts at Amherst in 1995. He is the Henson Professor of Software Engineering in the Department of Computer Science and Engineering at the University of Nebraska-Lincoln. He worked for six years as a senior engineer with Intermetrics Inc., developing compilers and software for safety-critical embedded systems. His research interests include software analysis, verification, and testing. He has served as a program chair for the SPIN Workshop on Model Checking of Software (2001), the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (2002), the ACM SIGSOFT Symposium on Foundations of Software Engineering (2004), the ETAPS Conference on Fundamental Approaches to Software Engineering (2007), and the International Conference on Software Engineering (2008). He is a member of the IEEE Computer Society and an ACM distinguished scientist.

Jiangfan Shi received the BS degree in computer applications from the North China Institute of Technology and the MS degree in computer science from the University of Nebraska at Omaha. He is currently a PhD student in computer science at the University of Nebraska-Lincoln, where he is a member of the Laboratory for Empirically based Software Quality Research and Development (ESQuaReD). His research interests include software testing and verification. He is a student member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.