Greedy Combinatorial Test Case Generation Using Unsatisfiable Cores

Akihisa Yamada
University of Innsbruck, Austria
akihisa.yamada@uibk.ac.at

Armin Biere
Johannes Kepler University, Austria
biere@jku.at

Cyrille Artho, Takashi Kitamura, Eun-Hye Choi
National Institute of Advanced Industrial Science and Technology (AIST), Japan
{c.artho,t.kitamura,e.choi}@aist.go.jp

ABSTRACT

Combinatorial testing aims at covering the interactions of parameters in a system under test, while some combinations may be forbidden by given constraints (forbidden tuples).

In this paper, we illustrate that such forbidden tuples correspond to unsatisfiable cores, a widely understood notion in the SAT solving community. Based on this observation, we propose a technique to detect forbidden tuples lazily during a greedy test case generation, which significantly reduces the number of required SAT solving calls. We further reduce the amount of time spent in SAT solving by essentially ignoring constraints while constructing each test case, but then "amending" it to obtain a test case that satisfies the constraints, again using unsatisfiable cores. Finally, to complement a disturbance due to ignoring constraints, we implement an efficient approximative SAT checking function in the SAT solver Lingeling.

Through experiments we verify that our approach significantly improves the efficiency of constraint handling in our greedy combinatorial testing algorithm.

Keywords

Combinatorial testing, test case generation, SAT solving

CCS Concepts

•Software and its engineering → Software testing and debugging;

1. INTRODUCTION

Combinatorial testing (cf. [24]) aims at ensuring the quality of software testing by focusing on the interactions of parameters in a system under test (SUT), while at the same time reducing the number of test cases that has to be executed. It has been shown empirically [23] that a significant number of defects can be detected by *t-way testing*, which tests all t-way combinations of parameters at least once, where t is a relatively small number.

Constraint handling, mentioned already by Tatsumi [34] in the late '80s, remains as a challenging research topic in combinatorial testing [25]. To illustrate the concept, we take a simple web application example which is expected to work in various environments listed as follows:

Parameter	Values
CPU	Intel, AMD
OS	Windows, Linux, Mac
Browser	IE, Firefox, Safari

Combinatorial testing aims at covering all combinations of values, but not all of them are necessarily executable; e. g., we have the following constraints:

1. IE is available only for Windows.
2. Safari is available only for Mac.
3. Mac does not support AMD CPUs.

Thus one must take care of such combinations which cannot be executed, called *forbidden tuples*. In the above example, there are six forbidden tuples: {Linux, IE}, {Linux, Safari}, {AMD, Mac}, etc.

There is substantial work on combinatorial testing taking constraints and forbidden tuples into account, including *meta-heuristic* approaches [10, 17, 20, 27], *SAT-based* approaches [29, 36], and *greedy* approaches, which is further categorized into *one-test-at-a-time* (OTAT) approaches [9, 11, 10] and *in-parameter-order generalized* (IPOG) approaches [37, 38].

Meta-heuristic approaches and SAT-based approaches often generate test suites that are smaller than the greedy approaches, although they usually require more computation time (cf. [20, 36, 27]). Thus, these approaches are preferred in case the cost of test execution is high.

On the other hand, there are practical needs for quickly generating a test suite of reasonable size, while the size is not the primary concern. For instance, if test execution is automated, one might better start executing the test cases, instead of waiting for a sophisticated algorithm to find a smaller test suite. Also in the phase of test modeling, one might want to check how test cases look like for an unfinished test model, not expecting a highly optimized test suite that would be beneficial if it were to be executed.

ASE'16, September 03 - 07, 2016, Singapore, Singapore

ISBN 978-1-4503-3845-5/16/09…$15.00

DOI: http://dx.doi.org/10.1145/2970276.2970335

In the IPOG-based approach, Yu et al. [37] proposed an efficient constraint handling mechanism which made the approach practical in the presence of complex constraints. In more recent work, Yu et al. [38] significantly improved the efficiency of the approach, by developing a dedicated analysis of *minimal forbidden tuples*.

In the OTAT approach, on the other hand, such a significant progress towards efficient constraint handling has not yet been made [8]. There is both theoretical and practical interest in this approach: An ideal greedy OTAT algorithm—ignoring constraints—is shown to deliver a test suite whose size is logarithmic to the number of parameters in the input SUT model [9]. A similar result is known [7] for the more feasible *density algorithm* of this category. In addition, the nature of generating "one test at a time" can be beneficial since one can start test execution before the entire test suite has been generated.

In this paper, we introduce an efficient constraint handling technique for the OTAT algorithms.

The first challenge for efficient test suite generation is how to efficiently detect forbidden tuples. To this end, we exploit the information of *unsatisfiable cores*, a notion widely understood in the SAT community [5]. In essence, we point out that every forbidden tuple corresponds to an unsatisfiable core—more precisely, the *failed assumptions* in it. Using failed assumptions, which IPASIR[1]-compliant SAT solvers can provide, we propose a technique to *lazily* detect forbidden tuples during greedy test case construction.

The second challenge is that, due to the nature of OTAT algorithms, still a larger number of SAT solving calls is needed. We show that most of these SAT solving calls are not required to guarantee the termination of the algorithm, but are needed to ensure that the given constraints are satisfied by the generated test cases. We introduce a new technique to "amend" a test case—turn a test case that possibly violates the constraints into one that satisfies the constraints—again using failed assumptions. Then we show that we can omit most of the SAT solving calls without affecting the correctness of the overall algorithm.

Finally, this omission of SAT solving makes the greedy test case generation heuristic to be approximative, i. e., it can make a wrong choice that will later be amended. Hence, we propose reducing the chance of making such wrong choices by exploiting the internal reasoning of SAT solvers. We added a new API function in the SAT solver Lingeling [2] that instantly checks satisfiability but allows for the third answer "unknown". We experimentally show that this technique pays off in terms of the sizes of generated test suites, with a mild computational overhead.

In principle, the proposed constraint handling method is applicable to any algorithms that comply the OTAT framework of Bryce et al. [8]. We implement the method in our base OTAT algorithm that is inspired by PICT [11] and AETG [9], and experimentally show that the proposed constraint handling method delivers a significant improvement to the efficiency of the algorithm.

This paper is organized as follows: Section 2 defines combinatorial testing formally, and Section 3 describes the usage of embedded SAT solvers. Section 4 shows the greedy (base) variant of our algorithm, where we observe that forbidden tuples correspond to unsatisfiable cores, as described in Section 5. Section 6 uses unsatisfiable cores to amend test cases, and Section 7 uses an extension of our SAT solver Lingeling [2] to optimize this algorithm. Section 8 gives on overview of related work. The results of our experiments are shown in Section 9, and Section 10 concludes.

[1]IPASIR is the new incremental SAT solver interface used in the SAT Race/Competition 2015/2016.

2. COMBINATORIAL TESTING

We define several notions for combinatorial testing. First, we define a model of a system under test (SUT).

Definition 1. *An* SUT model *is a triple* $\langle P, V, \phi \rangle$ *of*

- *a finite set* P *of* parameters*,*
- *a family* $V = \{V_p\}_{p \in P}$ *that assigns each* $p \in P$ *a finite set* V_p *of* values*, and*
- *a boolean formula* ϕ *called the* SUT constraint*, whose atoms are pairs* $\langle p, v \rangle$ *of* $p \in P$ *and* $v \in V_p$*.*

Hereafter, instead of $\langle p, v \rangle$ we write $p.v$ or even v if no confusion arises.

A valid test case is a choice of values for parameters that satisfy the SUT constraint.

Definition 2 (test cases)**.** *Given SUT* $\langle P, V, \phi \rangle$*, a* test case *is a mapping* $\gamma : P \to \bigcup V$ *that satisfies the* domain constraint*:* $\gamma(p) \in V_p$ *for every* $p \in P$*. A test case is called* valid *if it satisfies* ϕ*; more precisely, the following assignment* $\widehat{\gamma}$ *satisfies* ϕ*.*

$$\widehat{\gamma}(p.v) := \begin{cases} \textsc{True} & \text{if } \gamma(p) = v \\ \textsc{False} & \text{otherwise} \end{cases}$$

We call a set of valid test cases a test suite*.*

Example 1. *Consider the web-application mentioned in the introduction. The SUT model* $\langle P, V, \phi \rangle$ *for this example consists of the following parameters and values:*

$$\begin{aligned} P &= \{\text{CPU}, \text{OS}, \text{Browser}\} \\ V_{\text{CPU}} &= \{\text{Intel}, \text{AMD}\} \\ V_{\text{OS}} &= \{\text{Windows}, \text{Linux}, \text{Mac}\} \\ V_{\text{Browser}} &= \{\text{IE}, \text{Firefox}, \text{Safari}\} \end{aligned}$$

Following convention, the size of this model is denoted as $2^1 3^2$*, meaning that there is one parameter with two values and two with three values. The constraint in the introduction are expressed by the following SUT constraint:*

$$\phi := (\text{IE} \Rightarrow \text{Windows}) \wedge (\text{Safari} \Rightarrow \text{Mac}) \wedge (\text{Mac} \Rightarrow \neg\text{AMD})$$

The following table shows a test suite for the SUT model consisting of seven valid test cases.

No.	CPU	OS	Browser
1	AMD	Windows	IE
2	Intel	Windows	Firefox
3	Intel	Linux	Firefox
4	Intel	Windows	IE
5	Intel	Mac	Safari
6	AMD	Linux	Firefox
7	Intel	Mac	Firefox

The observation supporting combinatorial testing is that faults are caused by the interaction of values of a few parameters. Such interactions are formalized as follows.

Definition 3 (tuples)**.** *Given SUT $\langle P, V, \phi \rangle$, a* parameter tuple *is a subset $\pi \subseteq P$ of parameters, and a* (value) tuple *over π is a mapping $\tau : \pi \to \bigcup V$ that satisfies the domain constraint. Here, π is denoted by $\mathsf{Dom}(\tau)$.*

We identify a tuple τ with the following set:

$$\tau = \{\, p.v \mid \tau(p) = v \text{ is defined} \,\}$$

A test case γ is also a tuple s. t. $\mathsf{Dom}(\gamma) = P$.

Definition 4 (covering tests)**.** *We say that a test case γ* covers *a tuple τ iff $\tau \subseteq \gamma$, i. e., value choices in γ meet τ. A tuple is* possible *iff a valid test case covers it; otherwise, it is* forbidden*. Given a set T of tuples, we say that a test suite Γ is T-*covering *iff every $\tau \in T$ is either forbidden or covered by some $\gamma \in \Gamma$.*

The *covering test problem* is to find a T-covering test suite. The terms *t-way* or *t-wise* testing and *covering arrays* (cf. [30]) refer to a subclass of the covering test problems, where T is the set of all value tuples of size t. The number t is called the *strength* of combinatorial testing.

Example 2. *The SUT model of Example 1 has 21 tuples of size two, where six out of them are forbidden. The test suite in Example 1 covers all the 15 possible tuples and thus is a 2-way covering test suite for the SUT model.*

3. SAT SOLVING

Satisfiability (SAT) solvers [5] are tools that, given a boolean formula in *conjunctive normal form (CNF)*, decide whether it is possible to instantiate the variables in the formula such that the formula evaluates to true.

More formally, consider a boolean formula ϕ over a set $\mathcal{X}$ of variables. An *assignment* is a mapping $\alpha : \mathcal{X} \to \{\text{True}, \text{False}\}$. It *satisfies* a formula ϕ iff ϕ evaluates to True after replacing every variable x in ϕ by $\alpha(x)$. A formula is *satisfiable* if it can be satisfied by some assignment, and is *unsatisfiable* otherwise.

When SAT solvers conclude unsatisfiability, they are typically able to output a (minimal) unsatisfiable core, which is defined as follows. Here, we consider a CNF also as a set of clauses.

Definition 5. *An* unsatisfiable core *of a CNF ϕ is a subset of ϕ which is unsatisfiable. An unsatisfiable core ρ is* minimal *if any proper subset of ρ is satisfiable.*

3.1 CDCL

The DPLL algorithm [12] with *conflict-driven clause learning (CDCL)* [28] is a de facto standard architecture of SAT solvers. The CDCL approach constitutes a backtrack-based search algorithm, which efficiently scans the search space of possible assignments for the variables of a given formula. Its basic procedure is to repeat (1) choosing a variable and assigning a truth value for it (*decision*) and (2) simplifying the formula based on decisions using *unit propagation*. During this procedure, it may detect a *conflict* of some combinations of decisions and propagated assignments. Then a cause of the conflict—a set of decisions that derives it—is analyzed, and a clause is learned to avoid the same conflict later on. After backtracking the learned clause forces the assignment of one of its variables to be flipped, which might trigger further propagation. During this procedure the algorithm also checks whether all variables have been assigned and no more propagations are pending. In this case it terminates indicating that the formula is satisfiable.

3.2 Incremental SAT Solving and Failed Assumptions

Incremental SAT solving facilitates checking satisfiability for a series of closely related formulas. It is particularly important [33, 13] in the context of *bounded model checking* [4]. State-of-the-art SAT solvers like Lingeling [3] implement the assumption-based incremental algorithm, as pioneered by the highly influential SAT solver MiniSAT [14].

Incremental SAT solvers remember the current state and do not just exit after checking the satisfiability of one input formula. Besides asserting clauses that will be valid in the later satisfiability checks, incremental SAT solvers accept *assumption literals* [14], which are used as forced decision and are only valid during the next incremental satisfiability check, thus abandoned in later checks.

When an incremental SAT solver derives unsatisfiability, it is particularly important to know which assumption literals are a cause of unsatisfiability—in other words, constitute an unsatisfiable core. Such literals are called *failed assumptions*.

The interface of an incremental SAT solver is expressed in an object-oriented notation as follows, which is also compatible with the IPASIR interface.

```
class solver {
    literal      newVar();
    void         assert(clause C);
    void         assume(literal l);
    bool         check();
    assignment   model; // refers to a solution if exists
    list⟨literal⟩ failed_assumptions;
};
```

4. BASE GREEDY ALGORITHM

Before introducing constraint handling, we introduce our base OTAT algorithm without constraint handling, which is shown as Algorithm 1.

The algorithm works as follows: To generate one test case, the first step picks up a parameter tuple that has most uncovered tuples, and fixes those parameters to cover one of the uncovered tuples (lines 2–3). The second step greedily chooses a parameter and a value so that the number of newly covered tuples is maximized (lines 4–6). It may happen that fixing any single parameter/value will not increase the number of covered tuples, but fixing more than two will. In such a case, we apply the first step again to fix multiple parameters (lines 7–9).

The first step mimics that of PICT, while the second step is similar to AETG. The main difference from AETG is that we search all unfixed parameters for a value that maximizes the number of newly covered tuples, while AETG randomly chooses a parameter to be fixed and searches only for the best value for the parameter. Here, we do some clever computation in order to efficiently search all parameters. Due to this difference, our algorithm is deterministic (for tie breaking we use the deterministic random number generator of the standard C library) and produces a fairly small test suite without requiring multiple test case generation runs.

Algorithm 1: Basic Greedy Test Case Generation

Input: An SUT model $\langle P, V, \phi \rangle$ and a set T of tuples
Output: A T-covering test suite Γ

1 **while** $T \neq \emptyset$ **do**
2 choose $\tau \in T$ s. t. $\mathsf{Dom}(\tau)$ contains most uncovered tuples (as in PICT);
3 $\gamma \leftarrow \tau$; `// cover at least this tuple`
4 **while** *there are an unfixed parameter p and a value v s. t. $\gamma \cup \{p.v\}$ covers some uncovered tuples* **do**
5 choose such p and v that maximize the number of covered tuples;
6 $\gamma \leftarrow \gamma \cup \{p.v\}$;
7 **if** *there is an uncovered tuple τ that may be covered by fixing more than two parameters in γ* **then**
8 choose such τ as in PICT; $\gamma \leftarrow \gamma \cup \tau$;
9 **go to** line 4;
10 Fix unfixed parameters in γ to arbitrary values;
11 $\Gamma \leftarrow \Gamma \cup \{\gamma\}$; `// add the new test case`
12 Remove from T the tuples covered by γ;

4.1 Naive Constraint Handling

Now we present a variant of Algorithm 1 with naive constraint handling using incremental SAT solving, as already proposed by Cohen et al. [10].

Consider an SUT model $\langle P, V, \phi \rangle$. To represent a test case γ as a SAT formula, we introduce a boolean variable $p.v$ for each $p \in P$ and $v \in V_p$, denoting $\gamma(p) = v$. Since $\gamma(p)$ must be uniquely defined, we impose the following constraint:[2]

$$\textsc{Unique} := \bigwedge_{p \in P} \Big(1 = \sum_{v \in V_p} p.v\Big)$$

In the following algorithms, *tool* is assumed to be a SAT solver instance on which $\textsc{Unique} \wedge \phi$ is asserted.

Algorithm 2 shows our first algorithm called "naive", which however utilizes incremental SAT solving. It works as follows: Before generating test cases, it first removes all forbidden tuples in the set T of target tuples that has to be covered (lines 4–5). Then, whenever it chooses a tuple or value, it checks if the choice does not violate the constraint ϕ with already fixed values in γ (lines 9 and 11).

The use of incremental SAT solving reduces the total cost of SAT solving [10]. Nevertheless, as we will see in the experimental section, Algorithm 2 is not efficient for large-scale test models due to the large number of required SAT solving calls. Hence in the next section, we try to improve efficiency by reducing the number of SAT solving calls.

5. FORBIDDEN TUPLES AS CORES

The most time-consuming part of Algorithm 2 consists of lines 4–5, where all forbidden tuples are removed *a priori*. Note that for t-way testing of an SUT model of size g^k, the number of tuples in T, that is, the number of SAT solving calls needed in this phase, sums up to $\mathcal{O}(g^t\, k^t)$.

Hence, as the first improvement to this naive algorithm, we propose to remove forbidden tuples *lazily*. The key observation is that a forbidden tuple corresponds to the set of failed assumptions in an unsatisfiable core.

[2] In order to encode the above formula into a CNF, we use the *ladder encoding* [18] for at-most-one constraints.

Algorithm 2: Naive Treatment of Constraints

1 **function** Possible(τ)
2 **foreach** $v \in \tau$ **do** *tool.assume*(v);
3 **return** *tool.check*();
4 **foreach** $\tau \in T$ **do** `// remove forbidden tuples`
5 **if** $\neg$Possible(τ) **then** $T \leftarrow T \setminus \{\tau\}$;
6 **while** $T \neq \emptyset$ **do** `// main loop`
7 choose $\tau \in T$ as in PICT;
8 $\gamma \leftarrow \tau$; `// cover at least this tuple`
9 **while** *there exist p and $v \in V_p$ s. t. $\gamma \cup \{p.v\}$ covers new tuples and* Possible($\gamma \cup \{p.v\}$) **do**
10 Choose such best p and v; $\gamma \leftarrow \gamma \cup \{p.v\}$;
11 **if** *there is $\tau \in T$ s. t.* Possible($\gamma \cup \tau$) **then**
12 choose such τ as in PICT; $\gamma \leftarrow \gamma \cup \tau$;
13 **go to** line 9;
14 At this point, Possible(γ) = True is ensured. Fix unfixed parameters in γ according to *tool.model*;
15 $\Gamma \leftarrow \Gamma \cup \{\gamma\}$; `// add the new test case`
16 Remove from T the tuples covered by γ;

Example 3. *Consider the SUT model of Example 1, and suppose that in a test case generation procedure,* Browser *has been fixed to* Safari *and* OS *has been fixed to* Mac. *Note that no conflict will arise at this point. Next, consider fixing* CPU *to* AMD. *This choice raises unsatisfiability in the* Possible *call in line 9 of Algorithm 2. The corresponding minimum unsatisfiable core is either of the following:*

$$\text{Mac} \wedge \text{AMD} \wedge (\text{Mac} \Rightarrow \neg\text{AMD}) \quad (1)$$

$$\text{Safari} \wedge \text{AMD} \wedge (\text{Safari} \Rightarrow \text{Mac}) \wedge (\text{Mac} \Rightarrow \neg\text{AMD}) \quad (2)$$

The set of failed assumptions in (1) *is* $\{\text{Mac}, \text{AMD}\}$*; this indicates that one cannot fix* OS *to* Mac *and* CPU *to* AMD *in a test case, i. e.,* $\{\text{Mac}, \text{AMD}\}$ *is a forbidden tuple. Similarly, core* (2) *indicates that* $\{\text{Safari}, \text{AMD}\}$ *is a forbidden tuple. In either case, we detect a forbidden tuple.*

Now we introduce Algorithm 3, called "lazy", which omits to remove the forbidden tuples *a priori*, but lazily removes them when SAT checks show unsatisfiability.

Algorithm 3: Lazy Removal of Forbidden Tuples

1 **function** Possible'(τ)
2 **foreach** $v \in \tau$ **do** *tool.assume*(v);
3 **if** *tool.check*() **then return** True;
4 **else**
5 $T \leftarrow \{\, \tau' \in T \mid \textit{tool.failed\_assumptions}() \not\subseteq \tau' \,\}$;
6 **return** False;
7 **while** *there is $\tau \in T$ s. t.* Possible'(τ) **do**
8 choose $\tau \in T$ as in PICT; $\gamma \leftarrow \tau$;
9 Do lines 9–13 of Algorithm 2, where Possible is replaced by Possible';

Function Possible'(τ) checks if the tuple (or equivalently, partial test case) τ is possible (line 3). If it is not the case, then the SAT solver provides a set of failed assumptions, such as $\{\text{Mac}, \text{AMD}\}$ in Example 3. We now know that tuples containing the failed assumptions are forbidden; hence we remove such tuples from T (line 5).

The correctness of Algorithm 3, i. e., that it terminates and generates a T-covering test suite, is easily proven.

Proposition 1. *Algorithm 3 is correct.*

Proof. In every iteration of the main loop, the first τ chosen in line 8 is either removed or covered by the newly added test case. Hence, the algorithm terminates as $|T|$ strictly decreases in each iteration. Clearly, all tuples in T are either forbidden or covered by the output test suite. □

In many examples, the lazy removal of forbidden tuples significantly improves the runtime of the algorithm. In certain cases, however, this approach still suffers from an excessive large number of SAT solving calls (see also the experimental section). This phenomenon is caused by the second occurrence of PICT-like value choices (line 11 in Algorithm 2); if the constraint is so strict that the current values in γ are not compatible with any other remaining tuples in T, then at line 11 we have to perform SAT checks for all the remaining tuples in T. Hence in the next section, we consider to even omit these SAT checks.

6. CORES FOR AMENDING TEST CASES

The termination argument of Proposition 1 shows that only the first SAT check in every iteration is crucial to achieve the termination of the algorithm. The remaining SAT checks are only necessary to ensure that intermediate value choices in γ never contradict the SUT constraint ϕ. However, γ need not always respect ϕ; it suffices if the final test cases added to the output in line 16 satisfy ϕ.

Hence, in all iterations we can omit the SAT checks except for the first one. Unfortunately, then the resulting test case γ is not guaranteed to be valid in line 14 anymore. To solve this problem, we propose a technique to "amend" such an invalid test case and turn it into a valid one, again using the information of failed assumptions.

In general, if an incremental SAT solver detects unsatisfiability, then one of the failed assumptions must be removed in order to satisfy the formula. In our application, this means that some of the value assignments must be abandoned. By repeatedly removing failed assumptions until the formula becomes satisfiable, we can derive a test case that satisfies the SUT constraint.

Example 4. *Consider again the SUT model of Example 1, and the following invalid test case:*

$$\gamma = \{\text{Safari}, \text{Mac}, \text{AMD}\}$$

Possible'(γ) *will return* False *with a set of failed assumptions, e. g.,* $\{\text{Mac}, \text{AMD}\}$. *This indicates that at least either* Mac *or* AMD *must be removed from the test case. Thus consider removing, e. g.,* AMD*:*

$$\gamma' = \{\text{Safari}, \text{Mac}\}$$

Possible'(γ') *returns* True*, with a satisfying assignment*

$$\gamma'' = \{\text{Safari}, \text{Mac}, \text{Intel}\}$$

which is a valid test case.

It is important to properly choose which value assignment to remove. Note that even the termination of the algorithm cannot be ensured if one removes the first assumptions that are made to cover the first tuple τ in line 8.

For this choice, we propose to remove the failed assumption that corresponds to the most recently chosen value assignment. The underlying observation for this choice is that later value assignments are decided depending on earlier choices; the earlier the value assignment is chosen, the more influential it is to the coverage of the test case.

This algorithm, which we call "amend", is shown in Algorithm 4.

Algorithm 4: Amending Test Cases

```
1  function Amend(γ) // make γ a valid test case
2      while ¬Possible'(γ) do
3          Identify p.v ∈ tool.failed_assumptions that is the
           most recently fixed;
4          γ ← γ \ {p.v};
5      Fix unfixed parameters in γ according to tool.model;
6      return γ;
7  while there is τ ∈ T s. t. Possible'(τ) do
8      choose τ ∈ T as in PICT; γ ← τ;
9      Do lines 4–9 of Algorithm 1 (ignoring constraints);
10     γ ← amend(γ);
11     Γ ← Γ ∪ {γ}; // add the new test case
12     Remove from T the tuples covered by γ;
```

Proposition 2. *Algorithm 4 is correct.*

Proof. The crucial point is that the assumptions made in line 8 to cover the first tuple τ will not be removed. Since the satisfiability of τ is ensured, any unsatisfiable core that is found later must contain a failed assumption that is added by later greedy choice. In the worst case, all the choices but τ may be removed, but still T strictly decreases by each iteration. □

As we see in the above proof, Algorithm 4 is correct; however, it may happen that many value choices in a test case, which are chosen to cover as many tuples as possible, are eventually abandoned. This may in some particular cases result in preferable randomness that eventually yields a smaller test suite, but in general disturbs the greedy heuristic and results in a larger test suite, especially when the considered SUT constraint is so strict that most value choices violate the constraint. Note also that the notion of strictness does not directly correspond to the size or complexity of the constraint.

7. AVOIDING WRONG CHOICES

Finally, we reduce the chance of making wrong choices by extending the SAT solver Lingeling with the following method:

bool *imply*(**literal** l);

Method *imply*(l) adds l as an assumption literal just like *assume*(l), but it additionally performs the unit propagation. If the unit propagation causes a conflict, then the method returns False and the SAT solver goes into the state where it derives unsatisfiability. Otherwise, l is added as an assumption literal.

Using this method we implement a function Maybe, which is an approximate variant of Possible'. Maybe(τ) tests if

τ is possible by dispatching the *imply* method on the back-end SAT solver, but will not perform the actual satisfiability check. That is, this function may fail to detect a conflict, even if τ is actually impossible. However, if it detects a conflict, then τ is indeed impossible and the lazy removal of forbidden tuples (Section 5) is performed.

The overall algorithm "imply", which leverages the new *imply* method, is presented in Algorithm 5.

Algorithm 5: Avoiding Wrong Choices

```
1  function MAYBE(τ)
2      foreach v ∈ τ do
3          if ¬tool.imply(v) then
4              T ← { τ | tool.failed_assumptions() ⊈ τ };
5              return FALSE;
6      return TRUE;
7  while there is τ ∈ T s. t. POSSIBLE'(τ) do
8      choose τ ∈ T as in PICT; γ ← τ;
9      Do lines 9–13 of Algorithm 2, where POSSIBLE is
       replaced by MAYBE;
10     γ ← AMEND(γ);
11     Γ ← Γ ∪ {γ}; // add the new test case
12     Remove from T the tuples covered by γ;
```

Since MAYBE is only approximate, it is not ensured that the test case γ is valid after all values are fixed. Thus, the application of the AMEND function from the previous section is a crucial step (line 10).

Proposition 3. *Algorithm 5 is correct.*

Proof. The reasoning is the same as Proposition 2. □

The estimation quality of MAYBE depends on how many clauses have been learned by the SAT solver. At the beginning of test suite generation, it may often fail to detect conflicts, but at this stage disturbance by AMEND should be small, since most tuples are yet to be covered and any test case will cover some of them. As more test cases are generated, MAYBE becomes more precise.

8. RELATED WORK

Here we recall previous work towards efficient constraint handling in combinatorial testing, and remark a few other uses of SAT solving for test case generation.

8.1 PICT

PICT [11] is a well-known combinatorial testing tool based on the OTAT approach. For constraint handling, it precomputes *all* forbidden tuples—regardless of t—and uses this information when greedily constructing a test case [37]. This approach is quite fast if the SUT constraint is *weak*, i. e., only few tuples are forbidden. However, it becomes intractable if the constraint is so strict that a great number of tuples are forbidden. Note again that the strictness of constraint does not correspond to the complexity or the size of the constraint. On the other hand, the efficiency of our approach is stable from the strictness of the SUT constraint.

8.2 AETG

Cohen et al. [10] pioneered the use of incremental SAT solving for constraint handling in their OTAT algorithm AETG [9]. All their algorithms however remove forbidden tuples before the actual test case generation phase, as in our "naive" algorithm. Hence, their AETG variants would also benefit from our ideas. We also did not follow their "may" and "must" analysis, which was introduced to prevent the back-end SAT solver from encountering unsatisfiability. In our usage, deriving unsatisfiability is the key to detecting forbidden tuples, and we have no reason to prevent it.

8.3 ACTS

ACTS [6] is another successful combinatorial testing tool, which is based on the IPOG algorithm [26]. Yu et al. [38] improved the constraint handling of the IPOG algorithm using the notion of *minimum forbidden tuples (MFTs)*, i. e., the forbidden tuples whose proper subsets are not forbidden. Their first algorithm precomputes all MFTs, and uses this information during the test case generation. Moreover, to relax the cost of computing all MFTs, which is significant if the constraint is complex [38], they further introduced an algorithm using *necessary forbidden tuples (NFTs)*, that computes forbidden tuples when it becomes necessary.

Their work largely inspired us, although we did not choose IPOG as our base greedy algorithm. The notion of MFTs is somewhat related to minimal unsatisfiable cores, i. e., unsatisfiable cores whose proper subsets are satisfiable. The idea of the NFT algorithm is also visible in our "lazy" algorithm. However, while minimal unsatisfiable cores are naturally obtained by CDCL SAT solving, computing MFTs is a hard optimization problem since one has to further minimize unsatisfiable cores in terms of the failed assumptions.

8.4 ICPL

In the context of *software product lines (SPLs)*, Johansen et al. [21] introduced the ICPL algorithm for t-way test suite generation for SPLs. ICPL also incorporates SAT solvers, and take a different approach to reduce the cost of precomputing forbidden tuples. Their algorithm generates t-way test suite by generating t'-way test suites for $t' = 1, 2, \ldots, t$. Using the information of t'-way forbidden tuples, they proposed an algorithm to efficiently compute $(t' + 1)$-way forbidden tuples.

Compared to their work, our approach does not require a particular phase for computing forbidden tuples, since they are provided for free by incremental SAT solvers as failed assumptions. We leave it for future work to experimentally compare our tool with ICPL; these tools assume different input formats.

8.5 Constraints in Other Approaches

Of course, there are efforts towards constraint handling in non-greedy algorithms. The SAT-based approach encodes entire test suite generation as a SAT formula [19], and hence constraints can be naturally encoded [29]. Also the use of SAT solving has been proposed for the simulated-annealing-based tool CASA [16]. Lin et al. [27] recently introduced the two-mode meta-heuristic approach in combinatorial testing, and their tool TCA produces notably small test suites.

The primary concern of these approaches are not on execution time, but on the size of test suites. Indeed, CASA, TCA, and the SAT-based test suite optimization function in

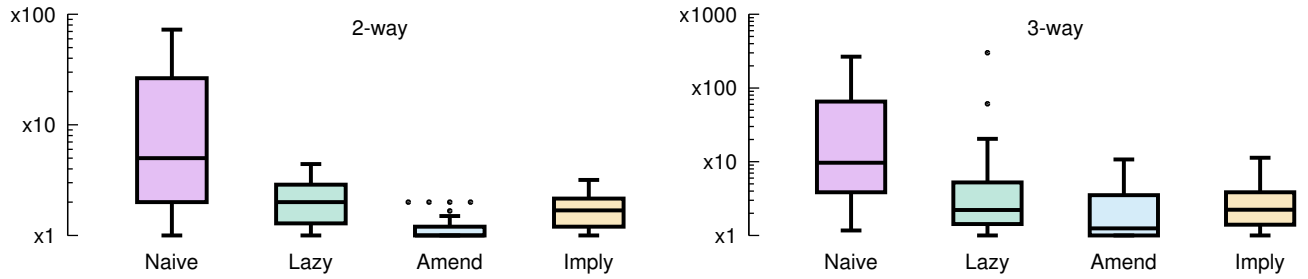

Figure 1: Comparison of the runtime of Algorithms 2–5. The ratio of the runtime over the best among all algorithms for each benchmark are plotted.

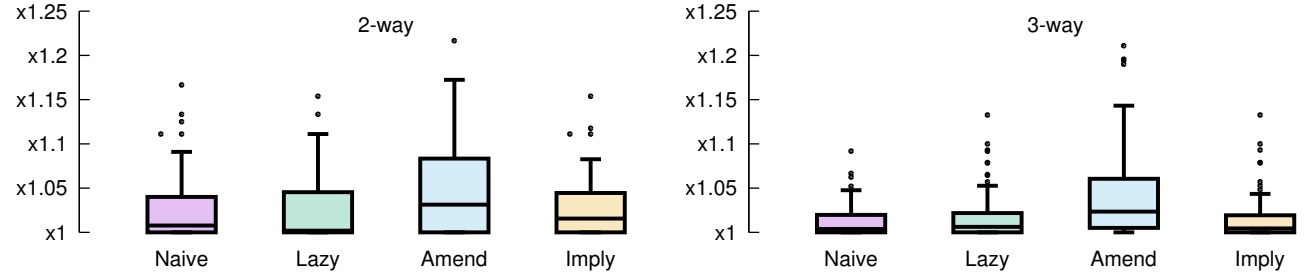

Figure 2: Comparison of Algorithms 2–5 in terms of the sizes of the generated test suites.

Calot [36] do not output a test suite even when they internally have one; they try to optimize it as far as possible, until certain termination conditions are met.

Nevertheless, we observe some cases where our work may provide a benefit to these approaches; e. g., Calot previously employed ACTS for constructing an initial test suite for optimization, which is now done efficiently inside Calot. TCA also employs a greedy algorithm for the initial test suite, and one of its "two modes" is a "greedy" mode, where we expect our technique can improve its efficiency.

Farchi et al. [15] proposed using unsatisfiable cores in the test modeling phase of combinatorial testing. Their test modeling tool FOCUS tests if tuples are possible or forbidden as expected. If a tuple is unexpectedly forbidden, then the tool analyzes the unsatisfiable core and indicates which clauses of the SUT constraint forbid the tuple. Thus users can effectively model an intended the SUT constraint. Our use of unsatisfiable cores makes a good contrast: Farchi et al. [15] consider assumed value choices are correct and fixes the *clauses* in an unsatisfiable core, while we consider the clauses are correct and fix the value choices.

8.6 SAT Solvers in Model-Based Testing

The *model-based testing (MBT)* (cf. [32]) considers more elaborated SUT models compared to combinatorial testing; namely, *states* of SUTs are considered. MBT tools aim at generating sequences of test cases which ensure a certain path-coverage criterion.

The use of incremental SAT solving in MBT is also proposed [1]. We expect that it is also interesting to use unsatisfiable cores to improve such SAT-based MBT tools; e. g., it might be able to efficiently detect "forbidden paths". We leave it for future work to explore to this direction.

9. EXPERIMENTS

We implemented Algorithms 2–5 in our tool Calot, and conducted experiments to investigate the following research questions.

RQ1 How efficient is the "lazy" algorithm compared to the "naive" one? How does it affect the sizes of test suites?

RQ2 How efficient is the "amend" algorithm compared to "lazy"? How does it affect the sizes of test suites?

RQ3 How much does the "imply" algorithm improve the sizes of test suites compared to "amend"? How does it affect the efficiency of the algorithm?

RQ4 How does the "imply" algorithm compare with other greedy test case generation tools?

As the benchmark set, we collected the following:

- the 35 benchmarks from Cohen et al. [10],
- the 20 industry applications from Segall et al. [31],
- the two industry applications from Yu et al. [38],
- the 18 industry applications from Kitamura et al. [22],
- and two applications from our industry collaborators.[3]

The experiments were run on a laptop with a 2.59GHz Intel Core i5-4310U processor and 4GB of RAM running Windows 10.

Figure 1 compares our four algorithms in terms of runtime for generating both 2-way and 3-way test suites, and Figure 2 compares the sizes of the generated test suites.

[3]One of our examples is available at https://staff.aist.go.jp/t.kitamura/dl/.

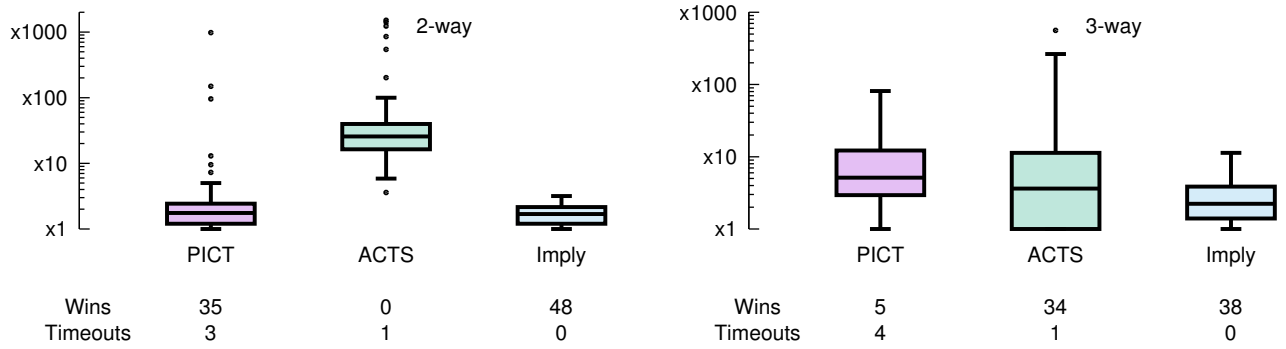

Figure 3: Comparing relative execution times with other tools.

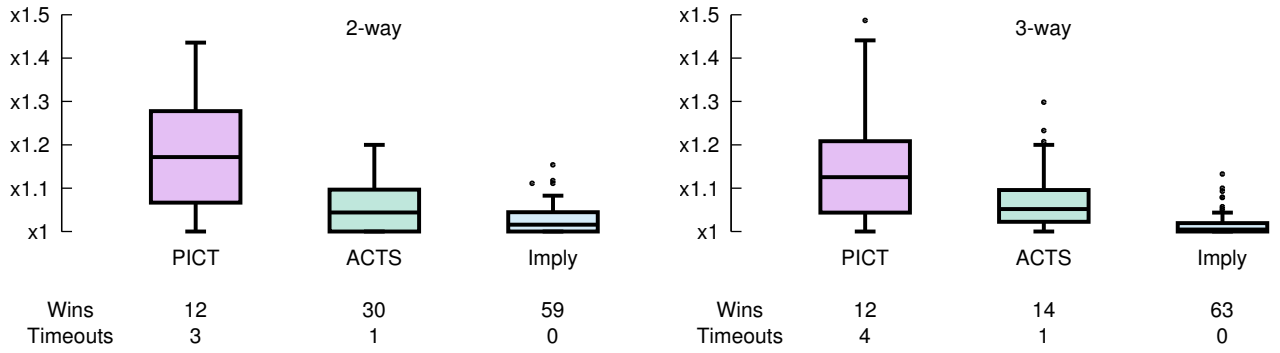

Figure 4: Comparing test suite sizes with other tools.

The box-plots show the distribution of the data: The box encompasses the first and third quartile of the distribution, with the middle line denoting the median value of the data. The "whiskers" are drawn at 1.5 times the interquartile range, to the data point closest to 1.5 times the distance between the median and the lower/upper quartile. Points outside that range are considered outliers and shown as a dot. To measure the validity of our claims, we also report the p-values in the Wilcoxon signed-rank test [35].

RQ1: Naive vs. Lazy

From Figure 1 we observe that "lazy" significantly improves the average execution time over "naive". The significance of the difference is $p < 0.0001$ for both 2- and 3-way cases. As explained in Section 5, however, there are a few examples in 3-way case where the runtime does not improve.

In terms of the sizes of test suites, from Figure 2 we observe only minor difference in 3-way case where "lazy" can be slightly worse than "naive", with $p \approx 0.1711$.

RQ2: Lazy vs. Amend

Here we measure the improvement due to amending invalid choices (see Section 6). In Figure 1, we observe further improvement in the efficiency of the "amend" algorithm over "lazy". The significance is $p < 0.0001$ for both 2- and 3-way cases. On the other hand, the sizes of generated test suites get noticeably worse (Figure 2), with $p < 0.0003$.

RQ3: Amend vs. Imply

Now we measure the improvement due to the new "imply" method (see Section 7). Our final algorithm "imply" improves the sizes of test suites over the previous "amend" algorithm with significance $p < 0.0002$, and apparently recovers to a similar result to "lazy". On the other hand, the overhead in runtime over "amend" is noticeable ($p < 0.0002$).

RQ4: Comparison with Other Greedy Tools

Finally, we compare our algorithms with the OTAT-based greedy tool PICT (version 3.3) and the IPOG-based greedy tool ACTS (version 2.93).

The results are shown in Figures 3–5, and also summarised in Table 1. The scatter-plots in Figure 5 compare individual data points between two settings. A data point above (below) the diagonal implies a higher (lower) value for the baseline algorithm—the naive algorithm.

For a few benchmarks PICT and ACTS do not respond within a reasonable time, so we set 3600 seconds as timeout. In case of timeout, we assumed 3600 seconds as the runtime. We do not know the size of the output test suite in these cases; hence Figure 4 excludes the case of timeouts in the box-plots. Instead, the number of timeouts are reported. The figures also report the number of "wins", i. e., how often the tool achieved the best result among others. When counting wins, ties are counted as a win for all tied tools.

When comparing the size of test suites, from Figure 4 we clearly observe that our algorithm outperforms others. The

Table 1: Detailed comparison with other tools. For each categories the average (avr.), geometric mean (g.m.), and the number of wins are reported. Fields with '–' cannot be computed due to timeouts.

			2-way						3-way					
			imply		PICT		ACTS		imply		PICT		ACTS	
Source	#		size	time	size	time	size	time	size	time	size	time	size	time
Cohen et al. [10]	35	avr.	**34.9**	**0.3**	43.9	0.2	36.4	4.6	**209.0**	26.5	252.0	83.2	219.7	**5.6**
		g.m.	**32.5**	**0.2**	40.4	**0.2**	33.8	3.9	**174.6**	8.4	206.5	16.4	183.6	**3.2**
		wins	**32**	**19**	1	18	9	0	**33**	3	1	3	3	**29**
Segall et al. [31]	20	avr.	74.5	**0.1**	77.0	0.2	**73.9**	1.0	**668.0**	1.2	684.5	1.9	675.0	**1.1**
		g.m.	**37.1**	**0.1**	39.7	**0.1**	**37.1**	0.8	**163.9**	**0.3**	174.0	0.5	171.2	0.9
		wins	11	**13**	4	10	**12**	0	**15**	**14**	5	2	5	4
Yu et al. [38]	2	avr.	467.5	**1.5**	–	1800.7	**461.0**	369.5	**7112.5**	**225.9**	–	1805.4	8054.5	360.6
		g.m.	348.6	**0.9**	–	71.7	**344.8**	146.7	**3716.1**	**64.4**	–	196.4	4117.5	149.0
		wins	0	**2**	0	0	**2**	0	**2**	**2**	0	0	0	0
Kitamura et al. [22]	18	avr.	**74.7**	**0.1**	75.8	217.7	76.3	15.3	**559.8**	**3.5**	–	220.7	567.6	14.9
		g.m.	**22.7**	**0.1**	22.1	0.3	23.8	1.8	**71.6**	**0.2**	–	1.1	74.3	1.8
		wins	**14**	**12**	7	7	7	0	**11**	**17**	6	0	6	1
Company_A	1		**42.0**	**0.1**	44.0	3221.5	43.0	120.7	**126.0**	**0.8**	–	3600.0	128.0	97.7
Company_B	1		**81.0**	**0.2**	–	3600.0	–	3600.0	**369.0**	**0.8**	–	3600.0	–	3600.0

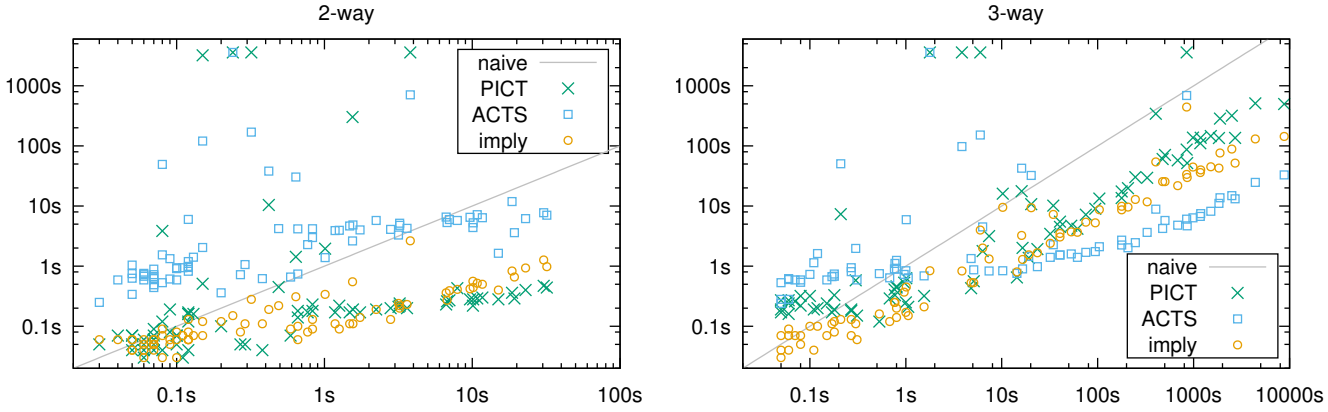

Figure 5: Comparing runtime with other tools. The vertical axis presents the runtime of each algorithm, while the horizontal axis presents the runtime the "naive" algorithm took for the same benchmark.

significance is $p < 0.0001$ for both 2- and 3-way, and for both PICT and ACTS.

When comparing runtime, from Figure 3 one might think that our "imply" is faster than the other tools. However, looking more detail in Figure 5 and Table 1 we see that the other tools perform quite well for many benchmarks. In particular, PICT is fast for some 2-way examples and ACTS is remarkably fast for many 3-way examples from Cohen et al. [10]. We conjecture that the efficiency of these tools depends on the weakness of the constraints.

10. CONCLUSION

In this paper, we have developed constraint handling techniques for one-test-at-a-time (OTAT) combinatorial test case generation algorithms, using the information of unsatisfiable cores. We implemented the proposed constraint handling methods in the OTAT algorithm of our test case generation tool Calot. Through experiments we verified that our techniques significantly improve the efficiency of test case generation without sacrificing the size of generated test suites, compared to constraint handling utilizing incremental SAT solvers naively. We also compared our tool with the greedy test case generation tools PICT and ACTS, and observed that our tool perform well in terms of both efficiency and the size of generated test suite, which are usually a trade-off with each other.

Limitation and Future Work.

Although Calot performed well in our experiments, if one ignores constraints, the IPOG algorithm of ACTS is remarkably faster than our base algorithm for large SUT models and higher-strength cases. This is explained by the fact that our base algorithm always has to compute the best parameters and values to fix, although this effort often results in a smaller test suite. Hence, we could only observe in some industry examples that the minimum forbidden tuple computation of Yu et al. [38] may become a significant overhead. We leave it for future work to implement our constraint handling approach in the IPOG algorithm to fairly compare our

constraint handling and the minimum forbidden tuple approach.

In principle, our constraint handling method can be immediately generalized to the OTAT framework by Bryce et al. [8]. We leave it for future work to develop such a framework where one can plug-in their own OTAT heuristics (such as the AETG or PICT heuristics), without being concerned about constraint handling.

11. ACKNOWLEDGMENTS

We would like to thank our two industry collaborators for allowing us to present experimental results on their applications. We thank the anonymous reviewers for their constructive and careful comments, which significantly improved the presentation of the paper. This work is in part supported by JST A-STEP grant AS2524001H.

12. REFERENCES

[1] P. Abad, N. Aguirre, V. Bengolea, D. Ciolek, M. F. Frias, J. Galeotti, T. Maibaum, M. Moscato, N. Rosner, and I. Vissani. Improving test generation under rich contracts by tight bounds and incremental sat solving. In *ICST 2013*, pages 21–30, 2013.

[2] A. Biere. Lingeling, Plingeling and Treengeling entering the SAT competition 2013. In *SAT Competition 2013*, pages 51–52, 2013.

[3] A. Biere. Yet another local search solver and Lingeling and friends entering the SAT Competition 2014. In *SAT Competition 2014*, volume B-2014-2 of *Department of Computer Science Series of Publications B*, pages 39–40. University of Helsinki, 2014.

[4] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS 1999*, volume 1579 of *LNCS*, pages 193–207, 1999.

[5] A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *FAIA*. IOS Press, February 2009.

[6] M. N. Borazjany, L. Yu, Y. Lei, R. Kacker, and R. Kuhn. Combinatorial testing of ACTS: A case study. In *ICST 2012*, pages 591–600, 2012.

[7] R. C. Bryce and C. J. Colbourn. The density algorithm for pairwise interaction testing. *Softw. Test., Verif. Reliab.*, 17:159–182, 2007.

[8] R. C. Bryce, C. J. Colbourn, and M. B. Cohen. A framework of greedy methods for constructing interaction test suites. In *ICSE 2005*, pages 146–155, 2005.

[9] 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.*, 23(7):437–444, 1997.

[10] M. B. Cohen, M. B. Dwyer, and J. Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Trans. Software Eng.*, 34(5):633–650, 2008.

[11] J. Czerwonka. Pairwise testing in real world. In *PNSQC 2006*, pages 419–430, 2006.

[12] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 4:394–397, 1962.

[13] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *Electr. Notes Theor. Comput. Sci.*, 89(4):543–560, 2003.

[14] N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT 2003*, volume 2919 of *LNCS*, pages 502–518, 2004.

[15] E. Farchi, I. Segall, and R. Tzoref-Brill. Using projections to debug large combinatorial models. In *IWCT 2013*, pages 311–320, 2013.

[16] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. An improved meta-heuristic search for constrained interaction testing. In *SSBSE 2009*, pages 13–22, 2009.

[17] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empir. Software Eng.*, 16: 61–102, 2011.

[18] I. P. Gent and P. Nightingale. A new encoding of alldifferent into SAT. In *International Workshop on Modelling and Reformulating Constraint Satisfaction*, pages 95–110, 2004.

[19] B. Hnich, S. D. Prestwich, E. Selensky, and B. M. Smith. Constraint models for the covering test problem. *Constraints*, 11(2-3):199–219, 2006.

[20] Y. Jia, M. B. Cohen, M. Harman, and J. Petke. Learning combinatorial interaction test generation strategies using hyperheuristic search. In *ICSE 2015*, pages 540–550, 2015.

[21] M. F. Johansen, Ø. Haugen, and F. Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *SPLC 2012, Volume 1*, pages 46–55, 2012.

[22] T. Kitamura, A. Yamada, G. Hatayama, C. Artho, E. Choi, T. B. N. Do, Y. Oiwa, and S. Sakuragi. Combinatorial testing for tree-structured test models with constraints. In *QRS 2015*, pages 141–150, 2015.

[23] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, Jr. Software fault interactions and implications for software testing. *IEEE Trans. Software Eng.*, 30(6):418–421, 2004.

[24] D. R. Kuhn, R. N. Kacker, and Y. Lei. *Introduction to Combinatorial Testing*. CRC press, 2013.

[25] D. R. Kuhn, R. Bryce, F. Duan, L. S. Ghandehari, Y. Lei, and R. N. Kacker. Chapter one – combinatorial testing: Theory and practice. volume 99 of *Advances in Computers*, pages 1–66. Elsevier, 2015.

[26] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. IPOG: A general strategy for t-way software testing. In *ECBS 2007*, pages 549–556, 2007.

[27] J. Lin, C. Luo, S. Cai, K. Su, D. Hao, and L. Zhang. TCA: An efficient two-mode meta-heuristic algorithm for combinatorial test generation. In *ASE 2015*, pages 494–505, 2015.

[28] J. Marques-Silva, I. Lynce, and S. Malik. Conflict-driven clause learning sat solvers. In A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, chapter 4, pages 131–153. IOS Press, 2009.

[29] T. Nanba, T. Tsuchiya, and T. Kikuno. Using satisfiability solving for pairwise testing in the presence of constraints. *IEICE Trans. Fundamentals*, E95-A(9), 2012.

[30] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Computing Surveys*, 43(2):11, 2011.

[31] I. Segall, R. Tzoref-Brill, and E. Farchi. Using binary decision diagrams for combinatorial test design. In *ISSTA 2011*, pages 254–264, 2011.

[32] M. Shafique and Y. Labiche. A systematic review of state-based test tools. *Int. J. Softw. Tools Technol. Transfer*, 17:59–76, 2015.

[33] O. Shtrichman. Pruning techniques for the SAT-based bounded model checking problem. In *CHARME 2001*, volume 2144 of *LNCS*, pages 58–70, 2001.

[34] K. Tatsumi. Test case design support system. In *Proc. International Conference on Quality Control (ICQC 1987)*, pages 615–620, 1987.

[35] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945. ISSN 00994987.

[36] A. Yamada, T. Kitamura, C. Artho, E. Choi, Y. Oiwa, and A. Biere. Optimization of combinatorial testing by incremental SAT solving. In *ICST 2015*, pages 1–10. IEEE, 2015.

[37] L. Yu, Y. Lei, M. Nourozborazjany, R. N. Kacker, and D. R. Kuhn. An efficient algorithm for constraint handling in combinatorial test generation. In *ICST 2013*, pages 242–251. IEEE, 2013.

[38] L. Yu, F. Duan, Y. Lei, R. N. Kacker, and D. R. Kuhn. Constraint handling in combinatorial test generation using forbidden tuples. In *IWCT 2015*, pages 1–9, 2015.