Dealing with Constraints in Boolean Expression Testing

Angelo Gargantini
Dipartimento di Ingegneria dell'Informazione e Metodi Matematici
Università di Bergamo - Italy
e-mail: angelo.gargantini@unibg.it

***Abstract*—When testing a Boolean expression, one should consider also the constraints among the variables contained in it. Constraints model interdependence among the conditions in the expressions. Only tests that satisfy the constraints. i.e. *valid* tests, are really useful and can be applied to test the expression. We present three ways to deal with such constraints: (1) ignoring them during test generation and removing invalid tests later, (2) including them in the expression as conjoint and again removing invalid tests later, and (3) considering them during the test generation process in order to generate only valid tests from the start. We introduce a general framework in which the three policies are implemented and compared over a set of Boolean expressions commonly used as benchmarks. Although the third policy requires a constraints solving technique for actual test generation, it presents several benefits: it generates smaller test suites and it may require less time for tests generation. Moreover, ignoring the constraints during test generation can reduce the fault detection capability of the tests.**

[Note: this version corrects several errors in the original paper - the author]

I. Introduction

Boolean expressions, i.e. terms that evaluate to true or false, are frequently found in logical predicates inside programs to model complex conditions under which some code is executed. They are commonly used as guards for conditional instructions and cycles. Also in model based testing, Boolean conditions play a very important role because they can be found as guards of transitions and actions. They constitute a critical part also because many typical programmer and designer errors result in faults in Boolean expressions. For instance, missing conditions are a typical fault due to an omission error, and error of omissions are one of most frequent error: Marick found that approximately half of the faults posted on usenet bug reports are faults of omission [22] (although not necessary faults in Boolean expressions). Boolean expressions are sometimes considered themselves *specifications*, and called in this way. Many specification formalisms such as the often used AND-OR tables (as those used in RSML [21] or in SCR [14]) can also be seen as Boolean expressions. In this work, we interchangeably use the terms Boolean specifications and expressions.

Testing Boolean expressions is a major topic both in program and model based testing and often referred as *logic testing*. Numerous testing criteria have been introduced in the past and they are continuously introduced. A survey on most logic based testing criteria is presented by [16] (12 testing criteria are considered) where the authors distinguish between semantics criteria (like MCDC and its variants) and syntactic criteria (like MUMCUT). They also discuss the test suite size and the fault detection capability of the testing criteria assessing their subsumption relationships. These criteria do not, however, deal explicitly with constraints. The constraints originate from interdependencies among the variables inside the Boolean specification under test. For instance, given the Boolean specification $(a \wedge b) \vee c$, if the designer knows that a implies b, then the constraint $\neg a \vee b$ should be eventually taken into account, since a test with a true and b false is useless. In this paper we identify three ways to deal with the constraints: (IGN) ignoring them during test generation and removing tests that do not satisfy the constraints later, (INC) including them in the expression as conjoint and again removing invalid tests later, and (VAL) considering them during the test generation process in order to generate only valid tests from the start. While the first two approaches are well known and used in the literature, the last approach has never been applied to logic testing. We compare the three approaches in terms of testing effort, test generation time, and fault detection capability. We study also the effect of some optimizations over the test suites. We found that the third approach has several advantages with respect the other two. Section II introduces the terminology we use and a technique based on SAT solving for test case generation. Section III presents the constraints, where they come from and the three ways they can be dealt with: (1) ignoring them (IGN), (2) including them in the expression under test (INC), and (3) considering them in order to generate only valid tests (VAL). Section IV reports the results we obtained from the comparison of the three approaches applied to a set of well known benchmarks.

II. Background

Terminology: Boolean expressions are those involving logical Boolean operators like AND, OR, and NOT (denoted by $\wedge, \vee, \neg, ..$) and those operands are *atomic* Boolean terms: atomic because they cannot be further decomposed in simpler Boolean expressions. We call the operands *inputs* or *variables* and use symbols like $x_1, x_2, \ldots$. The occurrence of an input in an expression is referred to as a *condition*. For example, the formula $x_1 \wedge x_2 \vee x_1$ contains two variables (x_1 and x_2), whereas the number of conditions is three (two x_1's and one x_2). If there are no restrictions on how operators and

conditions are joined together, we say that the expression is in a general form (GF), while if the expression is a disjunction of conjunctive conditions, then we say that is in disjunctive normal form (DNF). In this paper we consider GF Boolean expressions. In the context of testing Boolean predicates, a *test case* is a value assignment to every Boolean variable in the formula. A *test suite* simply is a set of test cases.

A. Test generation by SAT solving

In [1], [8] we presented an approach that allows the generation of tests by means of SMT and SAT solvers. It is a simplification of the test generation technique by model checking originally presented in [10], [9]. In this paper we assume that the specifications under test are Boolean expressions containing only Boolean variables (literals). Therefore, a testing criterion C is represented by a function that given a Boolean expression φ returns a set of predicates which must be covered. Each predicate represents a test goal and is called *test predicate*. We say that a $test$ satisfies or covers a test predicate tp iff $test$ is a model of tp, i.e. $test \models \mathsf{tp}$. A test predicate tp is said *infeasible* if there is no test that satisfies it, i.e. $\not\models \mathsf{tp}$.

Definition 1. Adequacy Given a testing criterion C and a Boolean expression φ, we say that a test suite *TS* is adequate to test φ according to C, iff $\forall \mathsf{tp} \in C(\varphi)\,(\not\models \mathsf{tp} \vee \exists test \in TS(test \models \mathsf{tp}))$

To discover if a test predicate is infeasible or find a test that covers it, a SAT/SMT solver can be used (assuming that the solver terminates otherwise it is not known whether the test predicate is infeasible or not). Note that infeasible test predicates consume computing resources without producing usable tests. The naive process of building a test for each test predicate can be improved by several technique: here we consider only monitoring and post reduction.

Monitoring: A test case explicitly generated for one test predicate may satisfy a number of further test predicates. Consequently, it is not strictly necessary with respect to achieving the complete coverage (i.e., satisfaction of all test predicates) to generate test cases for all test predicates. Instead, during the test generation process, each time a test case is generated the remaining uncovered test predicates can be checked against the new test case (i.e., they are *monitored* for satisfaction), and any satisfied test predicate can be omitted from test case generation because it is already covered.

Post reduction: The final test suite may contain tests which are not *necessary*. A test is not necessary if removing it from the test suite will lead to all test predicates still being covered (as in [13]). The problem of finding the optimal subset of the original test suite that still covers all the test goals is NP-hard, but can be efficiently solved by a simple greedy heuristic [4]. Post reduction can be performed in a negligible amount of time and it does not reduce the total number of test predicates covered by the test suite.

B. Fault-based Testing Criteria

In [8], we have presented fault based testing criteria which generate test predicates from Boolean expressions in disjunctive normal form (DNF) and whose tests are guaranteed to detect faults in specific fault classes (as in [19], [15]). This approach has been extended to Boolean expressions in General form (GF): although its detailed description is outside the scope of this paper, it can be summarized as follows. Several fault classes for Boolean expressions are considered (as in [17], [3]): given a specification φ, every fault class F builds all the possible faulty implementations φ'_i. By considering all the fault classes, the testing criteria compute all the the *test predicates* $\mathsf{tp}_i = \varphi \oplus \varphi'_i$, where $\oplus$ denotes the exclusive or. Finding the tests as models of the test predicates (i.e. $test_i \models \mathsf{tp}_i$) can be done by means of a SAT/SMT solver.

III. Constraints over variables in Boolean expressions

A Boolean expression often represents an abstract version of a real guard which may contain not only Boolean variables, but generic Boolean terms like relational expressions and method calls. In the abstract version, each term is simply substituted by a Boolean variable x_j. For this reason, some constraints between the variables are in place as well, for they model variable dependencies.

Example 2. Given the following C code fragment:

```
if((x<10 && y==5) || (x>20 && y!=5)){...
```

one would derive the Boolean expression $(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$. x_1 and x_3 cannot be both true at the same time (i.e. in the same test) and x_2 and x_4 are mutually exclusive (if one is true, then the other is false and vice-versa). Such constraints could be modeled as two Boolean predicates: $\delta_1 \equiv \neg(x_1 \wedge x_3)$ and $\delta_2 \equiv (x_2 \rightarrow \neg x_4) \wedge (x_4 \rightarrow \neg x_2) \equiv x_2 \oplus x_4$.

Example 3. The constraints sometimes generate from previous instructions or from other global constraints. For instance consider the following C code fragment:

```
y = -x;
if ( x >= 0 && y >= - 5) {...
```

One would abstract from the conditional guard the Boolean expression $x_1 \wedge x_2$. However, x_1 and x_2 are not independent since they cannot be both false: it cannot ever happen that x_1 is false (i.e. $x < 0$ holds) and x_2 is false (i.e. $y < -5$ holds: considering the assignment to y, it is equivalent to $x > 5$). This constraint could be modeled as Boolean expression: $\delta \equiv x_1 \vee x_2$.

It is clear that constraints pose a challenge during test generation and execution, because some combinations of variable values may be not satisfy such constraints. We say that a test is *valid* if it satisfies also the constraints. It is clear that in the end, only valid tests are really useful. We identified the following three main ways in the literature to deal with the constraints over the original specification.

1. IGN - Ignoring the constraints: the constraints are not considered during test case derivation. The testing criteria and the test generation algorithm are applied to the original formula. Tests that are not valid are later discarded. The main advantage of this approach, is that test generation algorithm can be applied without any modification, like no constraints were present. The disadvantages are that one could generate many more tests then necessary and that the elimination of the invalid tests from the test suite could reduce also its quality (in terms for example, of fault detection capability).

2. INC - Including the constraints as further conjoint in the original Boolean expression and the test generation algorithms are applied to the conjoint. In example 2, the Boolean expression would become $(x_1 \wedge x_2 \vee x_3 \wedge x_4) \wedge \delta_1 \wedge \delta_2$. The intent is clear: tests that are not valid will not exercise the decision or the guard represented by the expression. However, this approach has several shortcomings: it increases the size of the Boolean specification, and because most testing criteria require a number of tests which is proportional with the size of the expression, it also increases the test suite size. Moreover, this approach cannot distinguish between invalid tests from tests in which the original expression is tested false on purpose. Infact, a test that evaluates the conjoint to false, it may be invalid if it does not satisfy the constraints, but it may be valid in case it satisfies the constraints while falsifying the original Boolean specification.

3. VAL - Generating only valid tests: constraints are considered during test generation in order to generate only valid tests. In our framework, this means that a test must be a model of the test predicate and of the constraints as well, i.e. formally, given the constraints Δ , $test_i \models \Delta \wedge \mathsf{tp}_i$. The test generation must be able to deal with logic constraints and for this reason may require additional computational resources. Definition 1 can be modified as follows.

Definition 4. Adequacy in the presence of constraints. Given a testing criterion C, a Boolean expression φ, and the constraints Δ over it, we say that a test suite *TS* is adequate to test φ according to C, iff $\forall \mathsf{tp} \in C(\varphi)$ $(\not\models (\Delta \wedge \mathsf{tp}) \vee \exists test \in TS\,(test \models \Delta \wedge \mathsf{tp}))$

In the following, we will use IGN, INC, and VAL in order to refer to these three policies.

IGN and INC policies are commonly used in test generation for Boolean expressions [8], [23], [2], [18]. The VAL policy has never been applied to logic testing, but it is commonly used with test generation for programs using constraint solving techniques [11].

IGN and INC policies allow the test generation algorithm to consider only the structure of the expressions and the test generation process is greatly simplified. For this reason, testing criteria like MCDC and MUMCUT can still be applied by a simple enumeration algorithm. The VAL approach is feasible only if the test generation process is based on some technique capable of solving constraints, like those bases on Constraint Programming [6], [7] or SAT solving (like ours presented in this paper).

IV. Experiments

For experimentation, we consider the same set of predicates introduced by Weyuker et al. [23], who selected 13 Boolean conditions from the specification of TCAS II. They identified for 7 of them variable dependencies, and we restrict our experiments to those 7. Weyuker et al. [23] dealt with the constraints in two ways: ignoring and including them as conjoints. The specifications and therefore the same approach was taken by Chen et al. to introduce the testing criterion MUMCUT [2], by Kobayashi et al. [18] for evaluating the combinatorial and random/anti-random approaches to test generation, and by [15] to evaluate the Minimal-MUMCUT strategy. We found no testing criterion or technique that explicitly implements the third (VAL) approach for Boolean specifications.

We consider overall the 7 specifications with the constraints and the 3 policies: IGN and INC as in [23], and VAL.

Example 5. Consider for example the expression 20 used by [23] $\bar{e}f\bar{g}\bar{a}(bc + \bar{b}d)$[1]. Weyuker et al. discovered in the specification that conditions c and d could never be both true, so they transformed the original specification into specification 9 as $\overline{(\mathbf{cd})}\bar{e}f\bar{g}\bar{a}(bc + \bar{b}d)$. In our approach, specification 20 is used to test the IGN policy, while specification 9 is used to test the INC policy. Specification 20 is used also for VAL, together with the constraint $\neg(c \wedge d)$.

We generate test suites satisfying the fault-based criterion presented in Section II-B. In this way we are able to compare the fault detection capability of the test suites with ease. We have run the test generation algorithm for 20 times, and we report the average of the data obtained. As SAT solver, we use SAT4J [20] in this paper.

A. Testing effort

First of all, we wanted to measure the impact of the three policies on the effort necessary to build the final test suite. Table I reports the data we collected by generating the tests for the 7 specifications and the 3 policies. We found that:

1) ***test predicates*** (column *tp*): IGN and VAL have the same number of tps, so they present the same "complexity" for test generation and they both need in principle the same number of tests (assuming that the number of tps is a good measure of the complexity of the test generation and that on average a test covers an equal number of test goals). INC requires +60% of the test predicates of IGN and VAL: including the constraints in the expression increases the number of conditions in it, increases the number of possible faults, and therefore the number of test predicates.

2) ***infeasible*** test predicates (column *inf*): INC has the greatest number and the greatest ratio (23% of tps are infeasible) of them. Ignoring the constraints (IGN) has the beneficial effect of fewer infeasible test predicates and the lowest ratio of infeasibility (10%). Indeed, although VAL has the same number of test predicates as IGN, it has almost the double

[1]The '+' symbol represents the OR operator and adjacency between literals represents the AND operator.

	IGN					INC					VAL			
spec.	*#tp*	*#inf.*	*#tests*	*#val.*	*time*	*#tp*	*#inf.*	*#tests*	*#val.*	*time*	*#tp*	*#inf.*	*#tests/#val.*	*time*
1	229	15	20.8	11.6	1.9	462	107	28.9	15.6	1.9	229	15	13.5	0.6
2	426	77	27.0	16.0	3.2	919	143	40.0	24.0	3.3	426	77	20.1	1.5
3	1185	148	76.1	47.5	2.6	1478	467	68.0	47.0	7.8	1185	362	47.0	12.2
4	365	13	43.0	10.0	2.7	768	175	36.7	18.2	3.0	365	13	17.3	1.0
5	334	18	47.2	1.2	1.1	522	117	29.0	15.0	2.1	334	18	13.0	1.0
6	222	7	19.0	9.0	1.9	364	32	19.0	14.0	2.1	222	7	14.3	0.3
7	168	26	12.0	10.3	1.4	212	34	13.0	12.0	1.4	168	26	12.0	0.2
Σ	2929	304	245.1	105.6	14.6	4725	1075	234.6	145.8	21.6	2929	518	137.2	16.8

spec*: specification number,* **#tp***: number of test predicates that are generated,* **#inf.***: number of infeasible test predicates,* **#tests***: number of tests,* **#val.***: number of valid tests,* **time***: time taken to complete the generation in secs.*

Table I
TESTING EFFORT W.R.T. THE THREE POLICIES

of infeasible tps (+ 70%). This is an effect of the constraints, which make some combinations of variable values infeasible.

3) ***number of tests generated*** (column *tests*): IGN produces around the same number (+5%) of tests generated for INC: ignoring and including the constraints have a great impact over the number of test predicates, but no impact over the tests that must be generated to cover them. VAL produces the smallest test suite (- 41% w.r.t. INC).

4) ***number of valid tests***: in terms of valid tests, INC produces the greatest test suite, which contains only 6% more tests than those for VAL. Almost half of the tests generated by IGN are invalid and must be discarded: the final effect is that IGN has the smallest valid test suite.

5) ***time***: as expected INC requires much more time the IGN (around +50%). Although VAL has many more infeasible test predicates and it has to consider the constraints when generating the tests by SAT solving, overall the total time is around the same as the time required by IGN (+ 15%).

Overall, INC requires much more time but no so many more tests. IGN produces the smallest valid test suite. Limiting the generation to only valid tests (VAL) requires a little more than than ignoring them (IGN) but much less than including them (INC). However, IGN and INC could allow the use of lighter test generation techniques that consider only the structure of the expression under test.

B. Test suite quality

While smaller test suites are preferable in most cases, their quality must be assessed as well, since small test suite may have a limited fault detection capability. We can measure the *fault detection capability* in this way: given a test suite TS we eliminate from it invalid tests (i.e. tests that do not satisfy the constraints) and measure the number of faults f_{killed} still detected as the number of test predicates of the VAL specification covered by TS. Let $f_{VALfeasible}$ be the number of test predicates that are feasible in the VAL specification, we measure the fault detection capability as the ratio $f_{killed}/f_{VALfeasible}$. Table II reports the ratio depending on the policy and the optimizations used. Because the VAL test suites already cover all the feasible test predicates generated for VAL, their fault detection is already 100% and not reported in Table II.

In case A all the optimizations are used (as before in Table I); in cases B and C, described below, only one optimization at the time is used before removing the invalid test cases.

A. By applying monitoring, then post reduction, and then removing the invalid tests, we found that IGN detects only 71% of the faults on the average, while INC detects 90% of the faults. The relative quality of the IGN test suite w.r.t. the VAL test suite is also bad by considering that it requires 77% of the tests (105.6 vs. 137.2) to cover only 71% of the faults. So, it is true that IGN produces the smallest test suite, but it also provides a very limited fault detection capability. We believed that this was partially due to the use of the optimizations, which speed up the generation process but may decrease the quality of the test suite: by monitoring some tests are not generated and they may be useful when invalid tests are removed and post reduction possibly removes valid tests while keeping only tests which cover more test predicates but which may be invalid. For this reason, we performed some experiments without the use of optimizations.

B. Applying monitoring but not post reduction increases the quality of the IGN test suite but only marginally (from 71% to 73%) and increases the test size by 48%. For INC, avoiding the post reduction increases only the final size without increasing the fault detection capability.

C. Avoiding the use of monitoring and applying post reduction decreases the test suite size (by only 2 tests for IGN and by 1 for INC). This further reduction can be explained by the fact that without monitoring, post reduction is applied to a much bigger set of tests and it can reduce the set even more than the set already optimized by monitoring. This policy, however, decreases the quality of test suite only for IGN w.r.t. the use of monitoring (from 71% to 68%). However, completing the task requires around 50 times the time when monitoring is applied for IGN and 23 times for INC.

Avoiding post reduction improves marginally the test suite

	optimizations	policy	total size	total time (sec.)	fault ratio detection for specification (in %) *1*	*2*	*3*	*4*	*5*	*6*	*7*	Average
A	Mon./PostRed./Valid.	IGN	105.6	14.6	83	97	49	82	21	75	92	71
		INC	145.8	21.6	89	93	95	85	66	98	100	90
B	Mon./Valid.	IGN	155.8	26.6	89	90	32	78	35	89	100	73
		INC	195.5	21.6	89	93	95	85	66	98	100	90
C	PostRed./Valid.	IGN	104.3	730.5	83	86	34	73	24	76	98	68
		INC	143.8	506.0	89	93	95	85	66	98	100	90
D	Valid.	IGN	1962.6	730.5	89	93	47	79	40	92	100	77
		INC	2934.5	506.0	89	93	95	85	66	98	100	90
E	Valid./PostRed.	IGN	129.7	730.5	89	93	47	79	40	92	100	77
		INC	147.1	506.0	89	93	95	85	66	98	100	90

Table II
Optimizations and fault detection capability

quality. It seems that some test predicates are covered by tests that are filtered by post reduction before considering id they are valid. To validate this hypothesis, we have performed the following two experiments, in which we are sure that the optimizations do not influence the fault detection capability of the final test suite.

D: Not applying any optimization and then removing the invalid tests. This significantly increases the test suite size ($\times 19$ for IGN and $\times 20$ for INC), increases the quality of IGN (from 71% to 77%) but not for INC.

E: Avoiding the use of monitoring (in order to generate a test for every test predicate), removing the invalid tests, and only in the end applying post reduction. In this case the time and the fault detection capability is like when no optimization is applied, but the test suite size is reduced. For INC, fault detection is not increased and the size is almost identical w.r.t. the use of all the optimizations. For IGN, the final test suite is bigger then that obtained using all the optimizations (by around 23%) but also the fault detection capability is increased (from 71% to 77%).

We expected a better fault detection capability in cases D and E: if monitoring is not applied, then every feasible test predicate is covered by its own test, and if post reduction is applied only to valid tests, no invalids test is kept instead of a valid test. However, we found that some necessary tests (i.e. tests covering test predicates not covered by others) may still be invalid, as shown by the following example.

Example 6. Given an expression containing variables a, b and c, with the constraint $\delta = a \rightarrow \neg c$, the feasible test predicate $\mathtt{tp} = a \wedge b$ is covered by the test $a = \mathtt{true}$, $b = \mathtt{true}$, $c = \mathtt{true}$. However, this test is invalid because it does not satisfy δ. Removing it may leave the test predicate tp uncovered. This proves that some tests covering feasible test predicate may be invalid and yet necessary.

Overall, we can say that the use of optimizations does not decrease the test suite quality for INC and decreases only a little bit the quality for IGN. For IGN, the test suite with greatest fault detection capability is found when no optimization is applied at all, but this comes at the price of a much longer computation times. In any case, no test suite was able to detect 100% of the faults, with INC performing always better than IGN.

This means that the relatively low quality of the IGN test suite is due to the fact that ignoring the constraints causes the generation of many invalid tests which must be later discarded even if they would be necessary to have a good fault detection capability. IGN risks to produce small test suites of low quality.

Although the idea of including the constraints in the expression under test as conjoint has no clear motivation and seems to have several shortcomings, INC actually improves the final quality of the test suite and thanks to many optimizations the final test suite has similar size and it is obtained in a reasonable time.

Only VAL guarantees complete fault detection capability, with test suites in size between INC and IGN. This may be true when applying also other coverage criteria like MCDC.

V. Conclusions and Future Work

In this paper, we have studied three ways to deal with constraints in logic testing: ignoring them (IGN), including them as conjoint in the expression under test (INC), and considering them in the test generation process in order to obtain only valid tests (VAL). From our experiments, the VAL policy presents several benefits: reduced test suite size, complete fault detection, only valid tests are produced, and in a reduced time. However, it can be applied only if the test generation method allows it - in our case its implementation is straightforward because we use a logic solver. IGN allows the application of simpler test generation techniques but the quality of the final test suite can be compromised by removing invalid tests. INC is able to produce a good quality test suite, but it increases the number of conditions to be tested and this

requires much more resources (time and test predicates) than the other two.

The experimental results may depend on the particular testing criteria we have used, with other testing criteria (like the structural ones, as MCDC) the conclusion may differ. However, the presented insights hold in general: IGN produces some invalid tests which once removed may decrease the test quality, INC requires more resources because it increases the number of conditions in the expressions, and only VAL produces high quality test suites, but it requires a test generator method capable of constraints solving.

As future work, we plan to extend our study to combinatorial testing where constraints have a great importance as well [1]. Also in combinatorial testing, there exist several experiments showing that the INC policy is inadequate. For instance in [5], the authors conclude that: "This is strong evidence that constraint handling must be incorporated into CIT generation methods rather than added on as a post-processing phase". A similar conclusion is drawn by Grindal et alt. [12]: they compare four strategies to handle conflicts in combinatorial testing (using also a post reduction technique similar to that presented in this paper) and conclude that "the best method with respect to test suite size is to avoid selection of test cases with conflicts". They consider only test suite sizes and they do not, however, consider the fault detection capability when comparing the strategies. Also in program testing, constraints over the inputs, often modeled as preconditions, pose a great challenge in test generation. For instance, the generation of unit tests that exercise only valid method calls is of great interest. In the future, we plan to compare several policies to deal with input constraints in these area of testing too.

REFERENCES

[1] Andrea Calvagna and Angelo Gargantini. A formal logic approach to constrained combinatorial testing. *Journal of Automated Reasoning*, 45(4):331–358, 2010.

[2] T.Y. Chen, M.F. Lau, and Y.T. Yu. MUMCUT: A fault-based strategy for testing boolean specifications. In *Asia-Pacific Software Engineering Conference*, page 606, Los Alamitos, CA, USA, 1999. IEEE Computer Society.

[3] Zhenyu Chen, Tsong Yueh Chen, and Baowen Xu. A revisit of fault class hierarchies in general boolean specifications. *ACM Transactions on Software Engineering and Metholodogy*, 2011.

[4] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3), 1979.

[5] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Transactions on Software Engineering*, 34:633–650, 2008.

[6] Richard A. DeMillo and Jefferson A. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, Sep 1991.

[7] Jon Edvardsson and Mariam Kamkar. Analysis of the constraint solver in una based test data generation. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-9, pages 237–245, New York, NY, USA, 2001. ACM.

[8] G. Fraser and A. Gargantini. Generating minimal fault detecting test suites for boolean expressions. In *6th Workshop on Advances in Model Based Testing A-MOST 2010*. IEEE Computer Society, 2010.

[9] Angelo Gargantini. Using model checking to generate fault detecting tests. In *TAP'07: Proceedings of the 1st International Conference on Tests and Proofs*, volume 4454 of *Lecture Notes in Computer Science (LNCS)*, pages 189–206. Springer Verlag, 2007.

[10] Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. In Oscar Nierstrasz and Michel Lemoine, editors, *Proceedings of the 7th European Engineering Conference and the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1687 of *LNCS*, pages 6–10, Sep 1999.

[11] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using constraint solving techniques. In *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '98, pages 53–62, New York, NY, USA, 1998. ACM.

[12] Mats Grindal, Jeff Offutt, and Jonas Mellin. Managing conflicts when using combination strategies to test software. In *18th Australian Software Engineering Conference (ASWEC 2007), April 10-13, 2007, Melbourne, Australia*, pages 255–264. IEEE Computer Society, 2007.

[13] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A Methodology for Controlling the Size of a Test Suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, 1993.

[14] Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions of Software Engineering Methodology*, 5(3):231–261, 1996.

[15] G. K. Kaminski and P. Ammann. Using Logic Criterion Feasibility to Reduce Test Set Size While Guaranteeing Fault Detection. In *ICST'09: Proceedings of the 2nd International Conference on Software Testing Verification and Validation*, pages 356–365, Washington, DC, USA, April 1–4, 2009. IEEE Computer Society.

[16] Garrett Kaminski, Gregory Williams, and Paul Ammann. Reconciling perspectives of software logic testing. *Software Testing, Verification and Reliability*, 18(3):149–188, 2008.

[17] Kalpesh Kapoor and Jonathan P. Bowen. Test conditions for fault classes in Boolean specifications. *ACM Transactions on Software Engineering and Methodology*, 16(3):10, 2007.

[18] Noritaka Kobayashi, Tatsuhiro Tsuchiya, and Tohru Kikuno. Non-specification-based approaches to logic testing for software. *Information and Software Technology*, 44(2):113 – 121, 2002.

[19] Man Fai Lau and Yuen-Tak Yu. An extended fault class hierarchy for specification-based testing. *ACM Transactions on Software Engineering and Methodology*, 14(3):247–276, 2005.

[20] Daniel Le Berre and Anne Parrain. The SAT4J library, release 2.2, system description. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 7:59–64, 2010.

[21] Nancy G. Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.

[22] B. Marick. Two experiments in software testing. Technical Report UIUCDCS-R-90-1644, University of Illinois at Urbana-Champaign, 1990.

[23] Elaine Weyuker, Tarak Goradia, and Ashutosh Singh. Automatically generating test data from a Boolean specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, May 1994.