ISSN:0975-9646



# Test Suite Generation of t- way CIT with constraints for Web Based Application

**B. Vani**[#1], Dr. **R. Deepalakshmi**[#2]

[#1] *Assistant Professor, Department of CSE ,*
*Raja College of Engineering and Technology,Madurai, TamilNadu,INDIA.*

[#2] *Professor,Department of CSE ,*
*Velammal College of Engineering and Technology, Madurai, TamilNadu, INDIA.*

***Abstract*** **- Web based applications are increasing in importance as consumers use web for wide range of daily activities. Testing the web based applications as banking with large number of interactions is crucial. Combinatorial Interaction testing is a method that generates test suites incrementally using cum-variable strength strategy for testing. There are some unknown combinations that are impossible to occur due to the requirements set to the application termed as constraints. There arise practical concerns when adding constraints between combinations of input domain resulting in combinatorial explosion. This paper presents a new algorithm that features the construction of test suites to support expressive constraining over the input domain using predicates.**

***Keywords*** **- Interaction testing, Combinatorial Testing, Constraints, Test suite generation**

## I. INTRODUCTION

A combinatorial testing approach is a kind of functional testing technique consisting of exhaustively validating all combinations of size t of applications input values. It requires formal modeling of application features as input values. Modeling activities are expensive and time consuming. The tester models only the inputs and requires that they are sufficiently covered by tests. On the other hand unintended interactions between the input parameters can lead to incorrect behavior which may not be detected by traditional testing. In Particular Combinatorial Interaction Testing aims at generating the reduced-size test suite which covers all combination of input values with constraint support. Predicates and constrained covering array is used to generate test suites which forms the basis for combinatorial interaction testing.

## II. COVERING ARRAYS

A t-way CIT sample is a mathematical structure called covering array [1,2 ]. From the mathematical point of view, the problem of generating a minimal set of test suites covering all combinations of input values is equivalent to finding a covering array of strength t over a heterogeneous interaction. Covering arrays [3] are combinatorial structures which extend the notion of orthogonal arrays.

*DEFINITION 1.* An instance called an orthogonal array, OA(t, k, v) where every ordered subset occurs exactly once. In this case N is not used because the exact size of the array is always $v^t$.

OAs are tabular arrangement of symbols which satisfy certain combinatorial properties. It is the generalization of latin squares [4]. If OA Strength = 2 then every 2 columns contains all possible pairs of elements. Often OA matching the required combinatorial test structures does not exist. OAs don't support constraint among test settings and parameters.

*DEFINITION 2.* A covering array, CA(N; t, k, v), is an N × k array on v symbols with the property that every N ×t sub-array contains all ordered subsets of size t from the v symbols at least once.

An N x k array with the property that in every N x t sub - array, each t-tuple occurs at least λ times, where t is the strength of the coverage of interactions, k is the number of parameters (degree), and g = (g1; g2;…gk) is a vector of positive integers defining the number of values for each parameter.

*DEFINITION 3.* A mixed level covering array, MCA(N; t, k, $(v_1, v_2, ..., v_k)$), is an N × k array on v symbols, where $v = P_k\ i=1\ v_i$, with the following properties:

(1) Each column i ($1 \le i \le k$) contains only elements from a set $S_i$ of size $v_i$.

(2) The rows of each N × t sub-array cover all t-tuples of values from the t columns at least 1 time.

A shorthand notation is used to describe mixed level covering arrays [5] by combining equal entries in ($v_i : i \le 1 \le k$). For example three entries each equal to 2 can be written as $2^3$.

## III. CONSTRAINED COVERING ARRAYS

The presence of constraints demands new definition of proper CIT sample. Integral to this concept is whether t-set is consistent with constraints [6].

*DEFINITION 4.* Given a set of constraints C, a given t-set, s, is C-consistent if s is not forbidden by any combination of constraints in C.

This definition permits flexibility in defining the nature of constraints and how they combine to forbid combinations.

*DEFINITION 5.* A constrained-covering array, denoted CCA(N; t, k, v,C), is an N × k array on v symbols with constraints C, such that every N × t sub-array contains all ordered C-consistent subsets of size t from the v symbols at least once. We extend this definition to constrained mixed-level covering arrays CMCA(N; t, k, (v1, v2, ..., vk),C) in the natural way.

TABLE 1. SUMMARY OF CONSTRAINT HANDLING IN EXISTING ALGORITHMS/TOOLS

| Algorithm/Tool | Tool Category | Constraint Handling | Re-Implementable |
|---|---|---|---|
| AETG | AETG - Like Greedy | REMODEL | PARTIAL |
| DDA | AETG - Like Greedy | SIMPLE | YES |
| Whitch:CTS | Construction | EXPAND | NO |
| Whitch:TOFU | Unknown | EXPAND | NO |
| IPO | Greedy | NONE | --- |
| Test Cover | Construction | REMODEL | NO |
| Sim Annealing | Meta Heuristic | SIMPLE | YES |
| PICT | AETG - Like Greedy | REMODEL | PARTIAL |
| Constraint Solver | Constraint Solving | NONE | --- |

## IV. CONSTRAINT SUPPORT

A desired requirement of combinatorial interaction testing strategy is the ability to deal with complex constraints [7,8]. Although the presence of constraints reduces the size of combinatorial test suites it also makes test generation more challenging. The general problem of finding minimal test suite that satisfies t-wise interaction coverage is NP complete. If constraints are added on the input domain finding a single test suite that satisfies t-wise interaction coverage is NP complete.

There are already a few approaches dealing with constraints over the input domain [9, 10]. In order to deal with constraints some methods require remodeling the original specification. Some algorithms simply ignore constraints to post process the test suites, some others delete the combination of input that do not satisfy the constraints. The summary of constraint handling in the existing algorithms/tools is presented in table 1

## V. MODEL BY TEST PREDICATES

Theapproach formalizes combinatorial coverage by logic predicates [11, 12]. The preliminary definition of test and test suite is as follows. Given m input parameters, each ranging in its finite domain, a test is an assignment of values to each of the m parameters: $p_1 = v_1$, $p_2 = v_2$, ..., $p_m = v_m$or $<P_i = V_i>$. A test suite is a finite set of tests. The size of the test suite is the number of tests in it. To formalize CIT, express each of the combination of input as logic $T_{Pred}$ expression.

For Example :$p_1 = v_1$^ $p_2 = v_2$where p1 and p2 are two inputs or monitored variables of enumeration or boolean domain and v1 and v2 are two possible values of p1 and p2 respectively.

Similarly, t-wise coverage can be modeled by a set of test predicates, each of the type:

p1 = v1 ^ p2 = v2 ^ : : : ^ pt = vt$\equiv \bigwedge_{i=1}^{t} pi = vi$

where p1; p2 : : : pt are t input parameters and v1; v2, … , vt are their possible values. The t-wise coverage is represented by the set of test predicates that contains every possible combination of the t input variables with their values. Please note that to reach complete t-wise coverage this has to be true for each t-tuple of input parameters of the considered application.

To build the complete set of test predicates required for t-wise coverage of a model, employ a combinatorial enumeration algorithm, which simply takes every possible combination of t input variables and it assigns every possible value to them.

## VI. TEST GENERATION

The actual test generation [13] consists of finding a test that covers a given $T_{pred}$ i.e., a model for it. As long as constraint is not taken into account the $T_{pred}$ is a conjunction of atom of the form v=x and the model is trivial where simple algorithm is used. In order to support constraints the logical solver tools such as Symbolic Analysis Laboratory can better suit the task. The SAL Framework combines different tools of abstraction, solving and model checkers and used for test generation.

SAL offers a Bounded Model Checkers BMC and Symbolic Model Checkers SMC. A BMC transforms the model checking problem into a constraint satisfaction problem. A SMC uses Binary Decision Diagrams BDD to efficiently represent states, interaction relations and constraints among them.

In order to generate a test that covers a given combinatorial Tpred, SAL is asked to verify a trap property in the model containing monitored variables [14, 15]. The trap property states that $T_{pred}$ is never true or never($T_{pred}$). It enforces assignment of values falsifying trap property and satisfying $T_{pred.}$ The proposed approach is able to deal with temporal constraints and able to include state transition and interaction information that cannot be represented by SAT Solvers.

### A) IG-t-CCIT

The IG-t-CCIT is the Incremental Generation of t-way test suites with Constrained CIT with $T_{pred}$. The basic way to generate suitable test suite for t-wise coverage consist of executing the test predicate generator to generate predicate tree and order the $T_{pred}$ . Then collect all ordered test predicate from the list of candidate set and execute the SAL to remove the $T_{pred}$one by one until candidate set is empty. The SAL generates the test and passes on to Coverage Evaluator. The test plus Coverage information is passed on to test suite generator which works in two stages namely the expansion and contraction or reduction stage and provides the list of test suites.

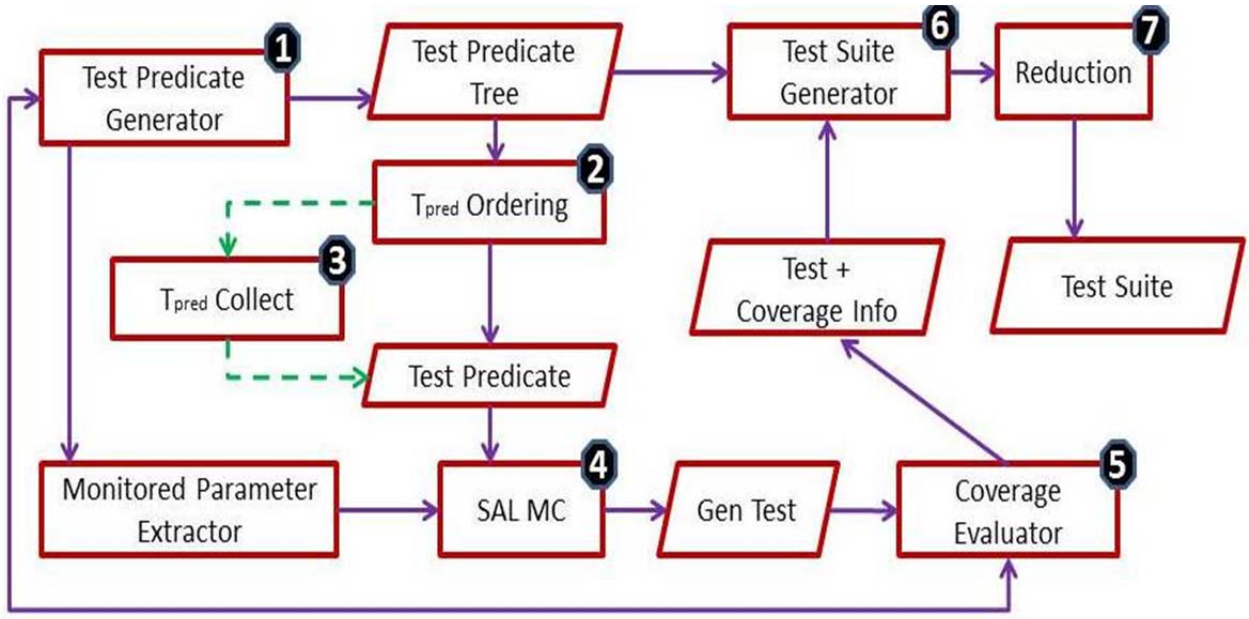


Figure 1 : Test Suite Generation Process by IG-t-CCIT Approach

B) $T_{pred}$Tree Construction

The Algorithm for the construction of $T_{pred}$Tree is as given in algorithm 1

**Algorithm 1 : $T_{pred}$Tree Construction**

**Input :** Monitored Parameters MP,
List of Constraints C
**Output :** Test Predicate Tree T

Begin
Generate list of tuples based on MP and store in CCA
X = get first tuple from CCA
Y= get other tuple from CCA
While X is not complete $T_{pred}$Tree
If X and Y can be fused and agree with all constraints C
Fuse X and Y to form new X
End if
Y = get other tuple from CCA
End While
Store X in T
Remove tuples covered by X from CCA
End

C) Monitoring

Every time a new test ts is added to the test suite, ts always covers as many as $\binom{m}{t}$t-wise test predicates, where m is the number of a system's input parameters and t is the strength of the covering array (t > 2 for combinatorial interaction testing). Checking which test predicates are covered by ts and remove them from the candidates leads to fewer calls to the model checker and possibly to smaller test suites. To enable monitoring, the tool detects if any additional test predicate $T_{pred}$ in the candidates is covered by ts by checking whether ts is a model of $T_{pred}$ (i.e. it satisfies $T_{pred}$) or not, and in the positive case it removes $T_{pred}$ from the candidates. Checking if a test is a model for a test predicate requires very limited computational effort [16]. This activity is performed by the Coverage Evaluator (stage 5 in Figure 1), which also computes the expected outputs as values for controlled parameters, if any.

D) $T_{pred}$Ordering

If monitoring is applied, the order in which the candidate test predicates are chosen and processed has a major impact on the size of the final test suite[17, 18]. In fact, each time a $T_{pred}$ is selected, a corresponding test case is generated, covering also other test predicates, which will be then removed from the candidate pool too. In fact, the more the candidate pool is reduced, the less the variety of test cases will be. Considering test predicates in the same order in which they are generated may lead to not optimal test suites. For this reason, insert an additional processing stage (stage 2 of Figure 1) in which the test predicates are ordered according to a user specified policy.

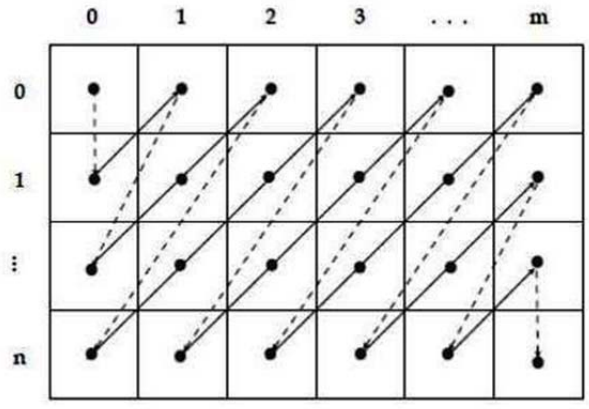


Figure 2 : A Schema of Anti-diagonal indexing of combination values

- Randomly – Choose the next predicate
- Order by Novelty – Choose from candidate pool according to well defined ordering criterion
- Anti-diagonal criterion - anti-diagonal criterion, which orders the test predicates such that no two consecutive $T_{pred} \equiv p1 = v1 \; and \; p2 = v2$ and $T_{pred}' \equiv p1' = v1' \; and \; p2' = v2'$ where $p1 = p1' and \; p2 = p2' will \; have \; v1 = v1' and \; v2 = v2'$.

E) Reduction

Monitoring can significantly reduce the size of a test suite, but a resulting test suite could still contain redundant tests[19, 20]. For Example, the last generated test might also cover several other test predicates $T_{pred}$previously covered by tests which may become useless. A smallest test suite is that in which each test predicate is covered by exactly one test case, but this very seldom happens: in most cases a test predicate will be covered by many tests creating possible redundancies. For this reason the analysis of the test suite is useful to further reduce it.

Test suite reduction (also known as test suite minimization) [21] is often applied in the context when one wants to find a subset of the tests that still satisfies given test goals.

## VII. ADDING CONSTRAINTS

Addition of constraints over the input is given by expressing them as axioms in the specification. In the bank mortgaging example, the assumption is that the customers has not availed the loan previously then check the type of customer and income and credit rating.

Axiom loan_notavailprev Over Home : (check Type of Customer)

Axiom Type of Cust = employee : (check creditrating implies excellent)

Axiom Type of cust = other : (check Income implies repayable)

To express constraint adopt the language of propositional logic with equality. For example the require constraint is translated by an implication. Note that also input domains must be taken intoaccount when checking axioms consistency [22]. Inconsistent axioms must be considered as a fault in the specification and this case must be (possibly automatically) detected andeliminated [23, 24].Even with consistent axioms, some (but not all) trap properties can be true: there is no test case that can satisfy the test predicate and the constraints. In this case define the test predicate as infeasible.

*DEFINITION 6.* Let $T_{pred}$ a test predicate, M the specification and $C_j$ the conjunction of all the axioms. If the axioms are consistent and the trap property for $T_{pred}$ is true i.e. $M \wedge C_j \Rightarrow \neg T_{pred}$, then $T_{pred}$ is infeasible. If $T_{pred}$ is the t-wise test predicate p1 =v1 ^ p2 = v2 … pt=vt then this combination of assignments is infeasible.

An infeasible combination of assignments represents a set of invalid test suites which contain such a combination are invalid. The proposed algorithm is able to detect an infeasible assignment since it can actually prove the trap property derived from it. In the bank mortgaging example consider M as loan_availed and $C_j$ as Income < Required the implication that is repayable property becomes false and combination is clearly infeasible. Stated Mathematically $M \wedge C_j \Rightarrow \neg T_{pred}$( Repayable = False).

Note that this infeasible combination is not explicitly listed in the constraints. Infeasible combinations represent implicit constraints. So every time we add the test predicate toconjoint of test predicates there is a need to check the consistency by considering the axioms also.

## VIII. RESULTS AND DISCUSSION

The proposed approach has been implemented in the ATGT. ATGT allows the tester to load an external file containing the user defined tests and goals [25, 26]. When the external file is loaded ATGT adds the user defined goals to set of test predicates to be covered. Then it adds the user defined tests and checks which $T_{pred}$ are satisfied by these tests. The proposed approach is applied to the web based banking application with different domain sizes using the constrained covering array. The approach is used to benchmark the size of generated test suite and assess the different test generation strategies.

TABLE 2 : TEST SUITE SIZE COMPARISON USING DIFFERENT ORDERING FOR UNCONSTRAINED MODELS

| Task | Size | No Collect | | Collect | | | Time (Secs) |
|---|---|---|---|---|---|---|---|
| | | Random | Novelty | Random | **Novelty** | AntiDiagonal | |
| CA1 | $3^3$ | 12 | 11 | 11 | **15** | 15 | **9.7** |
| CA2 | $4^3$ | 23 | 21 | 21 | **28** | 28 | **18.6** |
| CA3 | $5^3$ | 36 | 34 | 31 | **43** | 45 | **28.1** |
| CA4 | $6^3$ | 52 | 53 | 46 | **66** | 68 | **42.3** |
| CA5 | $7^3$ | 73 | 71 | 63 | **81** | 91 | **48.0** |

TABLE 3 : TEST SUITE SIZE COMPARISON USING DIFFERENT ORDERING FOR CONSTRAINED MODELS

| Task | Size | No Collect | | Collect | | | Time (Secs) |
|---|---|---|---|---|---|---|---|
| | | Random | Novelty | Random | **Novelty** | AntiDiagonal | |
| CCA1 | $3^3$ | 9 | 9 | 9 | **9** | 9 | **1.8** |
| CCA2 | $4^3$ | 17 | 18 | 17 | **17** | 18 | **4.7** |
| CCA3 | $5^3$ | 27 | 29 | 30 | **26** | 29 | **7.6** |
| CCA4 | $6^3$ | 41 | 41 | 41 | **38** | 44 | **11.2** |
| CCA5 | $7^3$ | 59 | 58 | 54 | **52** | 67 | **14.2** |

The comparison of the test suite size using unconstrained model and constrained model is given the table 2 and table 3. The results are worthwhile in terms of time and number of test suites. The Result chart shows the time difference for the generation of test suites for the unconstrained and constrained models.

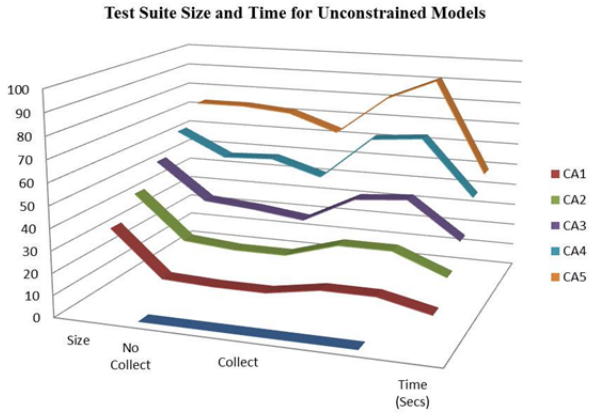


Figure 3: Test Suite Size and Time for Unconstrained Models

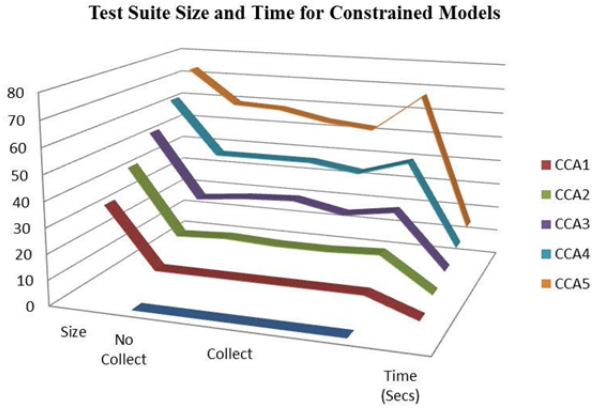


Figure 4: Test Suite Size and Time for Constrained Models

## IX. CONCLUSION

This paper presented the approach to t-way combinatorial test suite generation with support of constraints based on the predicates. The IG-t-CCIT has the ability to express complex constraints on the input domain in a compact and effective syntax as formal predicate expressions and axioms and able to generate optimized test suites along with user specific test goals. The constraint handling is done by predicates without having to remodel or expansion. It supports enumerative and Boolean types.

## REFERENCES

[1] Othman, Rozmie R., Kamal Z. Zamli, and Sharifah Mashita Syed Mohamad. "T-Way Testing Strategies: A Critical Survey and Analysis." International Journal of Digital Content Technology & its Applications 7.9 (2013).

[2] Khalsa, Sunint Kaur, and Yvan Labiche. "An orchestrated survey of available algorithms and tools for Combinatorial Testing." Carleton University, TR SCE-14-03, http://squall. sce. carleton. ca/pubs/tech_report/TR_SCE-14-03. pdf (2014).

[3] Johansen, Martin Fagereng, Øystein Haugen, and Franck Fleurey. "An algorithm for generating t-wise covering arrays from large feature models." Proceedings of the 16th International Software Product Line Conference-Volume 1. ACM, 2012.

[4] Zhang, Jian, Zhiqiang Zhang, and Feifei Ma. "Introduction to combinatorial testing." Automatic Generation of Combinatorial Test Data. Springer Berlin Heidelberg, 2014. 1-16.

[5] Gonzalez-Hernandez, Loreto. "New bounds for mixed covering arrays in t-way testing with uniform strength." Information and Software Technology 59 (2015): 17-32.

[6] Lopez-Herrejon, Roberto E., et al. "Towards a benchmark and a comparison framework for combinatorial interaction testing of software product lines." arXiv preprint arXiv:1401.5367 (2014).

[7] Liang, Xu, et al. "Combinatorial Test Case Suite Generation Based on Differential Evolution Algorithm." Journal of Software 9.6 (2014): 1479-1484.

[8] Mayo, Quentin, Ryan Michaels, and Renée Bryce. "Test Suite Reduction by Combinatorial-Based Coverage of Event Sequences." Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on. IEEE, 2014.

[9] Gao, Shiwei, et al. "An efficient algorithm for pairwise test case generation in presence of constraints." Systems and Informatics (ICSAI), 2014 2nd International Conference on. IEEE, 2014.

[10] Nie, Changhai, and Hareton Leung. "A survey of combinatorial testing." ACM Computing Surveys (CSUR) 43.2 (2011): 11.

[11] Mottu, J-M., et al. "Static analysis of model transformations for effective test generation." Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on. IEEE, 2012.

[12] Calvagna, Andrea, Angelo Gargantini, and Paolo Vavassori. "Combinatorial testing for feature models using citlab." Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on. IEEE, 2013.

[13] Fraser, Gordon, Andrea Arcuri, and Phil McMinn. "A Memetic Algorithm for whole test suite generation." Journal of Systems and Software (2014).

[14] Arcaini, Paolo, Angelo Gargantini, and Elvinia Riccobene. "How to Optimize the Use of SAT and SMT Solvers for Test Generation of Boolean Expressions." The Computer Journal (2015): bxv001.

[15] Segall, Itai, Rachel Tzoref-Brill, and Aviad Zlotnick. "Simplified modeling of combinatorial test spaces." Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on. IEEE, 2012.

[16] Arcaini, Paolo, Angelo Gargantini, and Paolo Vavassori. "Validation of models and tests for constrained combinatorial interaction testing." Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on. IEEE, 2014.

[17] Gargantini, Angelo, and Gordon Fraser. "Generating minimal fault detecting test suites for general boolean specifications." Information and Software Technology 53.11 (2011): 1263-1273.

[18] Pachauri, Ankur, and Gursaran Srivastava. "Automated test data generation for branch testing using genetic algorithm: An improved approach using branch ordering, memory and elitism." Journal of Systems and Software 86.5 (2013): 1191-1208.

[19] Hao, Dan, et al. "On-demand test suite reduction." Proceedings of the 34th International Conference on Software Engineering. IEEE Press, 2012.

[20] Coutinho, A. E. V. B., et al. "Test suite reduction based on similarity of test cases." 7st Brazilian workshop on systematic and automated software testing—CBSoft 2013. 2013.

[21] Güttinger, Dennis, et al. "Variations over Test Suite Reduction." Testing Software and Systems. Springer Berlin Heidelberg, 2013. 149-163.

[22] Abrusci, V. Michele. "On Hilbert's Axiomatics of Propositional Logic." Perspectives on Science 22.1 (2014): 115-132.

[23] Loveland, Donald W. Automated Theorem Proving: a logical basis. Vol. 1. Elsevier, 2014.

[24] Horvitz, Eric J. "Reasoning about beliefs and actions under computational resource constraints." arXiv preprint arXiv:1304.2759 (2013).

[25] Bolis, Francesco, et al. "Model-driven testing for web applications using abstract state machines." Current Trends in Web Engineering. Springer Berlin Heidelberg, 2012. 71-78.

[26] Yu, Linbin, et al. "Acts: A combinatorial test generation tool." Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on. IEEE, 2013.