Validation of Models and Tests for Constrained Combinatorial Interaction Testing

Paolo Arcaini
Dipartimento di Ingegneria
Università degli Studi di Bergamo, Italy
Email: paolo.arcaini@unibg.it

Angelo Gargantini
Dipartimento di Ingegneria
Università degli Studi di Bergamo, Italy
Email: angelo.gargantini@unibg.it

Paolo Vavassori
Dipartimento di Ingegneria
Università degli Studi di Bergamo, Italy
Email: paolo.vavassori@unibg.it

***Abstract*—In Combinatorial Interaction Testing, models specify a set of parameters with associated domains, and some constraints over the parameters. Test generation tools produce, starting from these models, test suites for achieving some given coverage criteria. The validation of both the models and the produced test suites is a worthwhile activity. Validating the models permits to early discover possible defects in them, to feed the test generations tools with *good* inputs and, possibly, to improve the quality of the testing process. Validating the produced test suites, instead, permits to check if the test generation tools are correct and to judge their quality. This paper proposes to validate the models by checking that the constraints are consistent, that there is no constraint implied by the other constraints, and that the parameters and their values are really necessary. The proposed test suite validation, instead, consists in checking that the tests respect the type definitions and the constraints, that all the test requirements are covered, and that all the tests in the test suite are valid and necessary. For every error we propose a possible technique able to identify the potential causes and to suggest fixes for those problems. Experiments show that the targeted defects are widespread both in benchmark and real-life models.**

I. Introduction

System validation is an essential activity of any development process since it permits detecting faults as early as possible. Model validation should precede the application of more expensive and accurate verification methods, that should be applied only when a designer has enough confidence that the specification really reflects the user perceptions. In case of test generation, models should be validated before tests are generated, otherwise the generated tests may be useless or even wrong if models they are generated from are not valid. Also test cases, especially when their generation is obtained using experimental tools and their instantiation can be costly, must be validated.

Among validation techniques, *model review*, also known as *model walk-through* or *model inspection*, allows to critically examine modeling efforts to determine if a model not only fulfills the intended requirements, but also is of sufficient quality to be easy to develop, maintain, and enhance. This process should, therefore, assure a certain degree of quality. When model reviews are performed properly, they can have a big payoff because they allow to identify defects early in the system development, reducing the cost of fixing them. Usually model review, which comes from the code-review idea, is performed by a group of external qualified people, often both technical staff and project stakeholders, who meet together to evaluate models and documents. The model inspection can be extended to test cases as well: the testers could manually check that the tests cover all the desired testing requirements.

A weak aspect of the review process is that it is usually done by hand. This requires a great effort that might be tremendously reduced if performed in an automatic way by systematically checking specifications for known vulnerabilities or defects. The question is *what* to check on and *how* to automatically check the model. In other words, it is necessary to identify classes of faults and defects to check, and to establish a process by which to detect such deficiencies in the model. If these faults are expressed in terms of formal statements, these can be assumed as a sort of "measure" of the *model quality assurance*. A tool is also necessary to make the process automatic. It would work as model advisor to check a model for conditions and configuration settings that can result in inaccurate or under-/over-specified behavior of the system that the model represents. The same applies to generated test suites: if the testing requirements are formally stated, they can be checked against the test suites in order to assure that they are actually achieved.

In this paper we focus on constrained combinatorial models and combinatorial tests generated from them. These models simply contain a set of parameters with their domains, together with some constraints among such parameters. They are classically validated by hand by the domain experts, who are able to judge if the constrained combinatorial models actually capture the problem to be tested. In this paper we focus on some properties that any combinatorial model should have, regardless the system it models. For instance, the constraints must not contradict each other, otherwise no suitable solution can exist. Moreover, we also focus on the validation of tests generated from such models. The activity of test generation is normally performed by suitable tools, which should guarantee that the test suites are valid, i.e., they satisfy all the testing requirements. However, sometimes tests could be (partially) inserted by hand, or the tools may contain faults that cause the generation of wrong tests or of incomplete test suites. Moreover, the instantiation of combinatorial testing is often a human activity that requires a great amount of time and resources, and wrong tests may cause serious problems. Consider, for instance, a combinatorial problem in which a certain configuration is forbidden because it leads the system to a dangerous state. In these cases, performing a validation process of the tests is worthwhile. Moreover, this technique can also be exploited to validate test generation tools.

The paper is organized as follows. Section II gives some basic definitions about combinatorial models, describes how satisfiability solvers can be used for combinatorial problems, and introduces a general description of the qualities expected from combinatorial models and tests. Desired properties of combinatorial models and tests are presented in Sections III and IV, together with the techniques to prove them. Section V describes some experiments done on benchmark case studies. Section VI presents some related work, and Section VII concludes the paper by presenting the settings in which the use of the proposed validation framework may be useful.

II. Basic definitions

A. *Combinatorial models and* CitLab

Combinatorial Interaction Testing (CIT), often called simply combinatorial testing or combinatorial testing design, aims at testing the software or the system with selected combinations of parameters values (inputs). There exist several tools and techniques for CIT. Good surveys on the ongoing research in CIT can be found in [21], [14], while an introduction on CIT and its efficacy in practice can be found in [17]. We assume in this paper that the reader is familiar with the CIT in general. A model for a combinatorial problem consists in several parameters (at least 2) which take values in their domains. In most configurable systems, constraints or dependencies exist between parameters. Constraints may be introduced for several reasons, for example, to model inconsistencies between certain hardware components, limitations of the possible system configurations, or simply design choices [9]. Constraints were first described as being important to combinatorial testing in [8] and were introduced in the AETG system. In our approach, tests that do not satisfy the constraints are considered *invalid* and do not need to be produced.

In this paper we assume that the models are specified using CitLab [12], [5]. CitLab is a framework for combinatorial testing which provides a rich abstract language with a precise formal semantics for specifying combinatorial problems, an eclipse-based editor with a rich set of features (like syntax highlighting, autocompletion, and outline view), and a Java APIs library which includes utility methods for generating all the test requirements for a combinatorial coverage of a given strength. CitLab does not have its own test generators, but it relies on other off-the-shelf tools (like ACTS[1] and CASA[2]).

In CitLab parameters and constraints are given in a unique file that contains the whole model. In CitLab, to formally describe a combinatorial problem, the user has to identify at least 2 parameters and their possible values. We call $P = \{p_1, \dots, p_m\}$ the set of parameters. Every parameter p_i assumes values in the domain $D_i = \{v^i_1, \dots, v^i_{o_i}\}$. Every

[1]http://csrc.nist.gov/groups/SNS/acts/

[2]http://cse.unl.edu/~citportal/

```
Model Phone
Definitions:
    Number threshold 27;
end
Types:
    EnumerativeType cameraType {1MP 2MP NOC};
end
Parameters:
    Range textLines [ 25 .. 30 ];
    Boolean emailViewer;
    Enumerative display {16MC 8MC BW};
    Enumerative rearCamera type cameraType;
    Enumerative frontCamera type cameraType;
end
Constraints:
    # emailViewer=true implies textLines >= threshold #
end
```

Figure 1. A smartphone example

parameter has its name (it can have also a type with its own name) and every enumerative value has an explicit name. We identify with $C = \{c_1, \dots, c_n\}$ the set of constraints.

CitLab adopts the language of propositional logic with equality and arithmetic to express constraints. To be more precise, we use propositional calculus, enriched with the arithmetic over the integers and enumerative symbols. As operators, we admit the use of equality and inequality for any variable, the usual boolean operators for boolean terms, and the relational and arithmetic operators for numeric terms. CitLab supports also seeds and test goals. However, they are supported by few tools and they are not considered in this work.

Fig. 1 reports the CitLab input domain model of a simple smartphone product line containing 5 parameters. The number of lines of the display (`textLines`) must be between 25 and 30; the phone can have an email viewer (`emailViewer`); the `display` can have 16 million colors or 8 million colors (16MC and 8MC), or be in black and white (BW); the rear and front cameras (`rearCamera` and `frontCamera`) can have 1 or 2 megapixels (1MP and 2MP) or be not present (NOC). In the **Definitions** section, it is possible to define constants to be used in formulas: for example, the constant `threshold` indicates the minimum number of lines of the display required by the email viewer. In the **Constraints** section a constraint specifies that, if the phone has the email viewer, the number of text lines of the display must be at least 27.

B. *Using Logics and SAT/SMT solvers for CIT problems*

In this paper, in order to analyze combinatorial models and tests, we adopt a logic-based approach.

Definition 1. Test A test $t = (v_1, \dots, v_m)$ is an assignment of values to all the parameters of the combinatorial problem that respects the type definitions, i.e., $\forall i \in \{1, \dots, m\} \colon v_i \in D_i$. Let $D = D_1 \times \dots \times D_m$ be the domain of the tests, i.e., $t \in D$.

Definition 2. A test t is a model for a formula φ if it makes the formula φ true, formally $t \models \varphi$.

Definition 3. Constraint A constraint c_i is a formula over some parameters of the combinatorial problem.

Definition 4. Test validity A test t is *valid*, if t is a model of the constraints, i.e., $t \models \bigwedge_{i=1}^{n} c_i$.

Given a formula φ over the parameters in P, there are several decision problems regarding the truth evaluation of φ.

Definition 5. Satisfiability A formula φ is satisfiable if there is model for it. Formally, $\exists t \in D\colon t \models \varphi$.

Satisfiability can be proved with a SAT/SMT solver. There are some formulas that are always true, regardless the test.

Definition 6. Validity A formula φ is valid if and only if it is true under every interpretation, i.e., every possible assignment to the parameters makes φ true. Formally, $\forall t \in D\colon t \models \varphi$, or briefly $\models \varphi$.

To prove validity one can use a satisfiability solver, thanks to the following theorem.

Theorem 7. *A formula φ is valid if $\neg\varphi$ is not satisfiable.*

In combinatorial testing, test requirements are given as a set of tuples which are to be covered by the tests. Each tuple can be represented as a formula in a straightforward way by a conjunction of equalities. For instance, the pair $(p_i = v_j, p_k = v_h)$ can be represented as $p_i = v_j \wedge p_k = v_h$. A test *covers* a test requirement, thus a tuple *tp*, if it makes *tp* true. A satisfiability solver can be used also to check if a test requirement is feasible.

Definition 8. A test requirement *tp* is feasible if $tp \wedge \bigwedge_{i=1}^{n} c_i$ is satisfabile.

In this paper we exploit, whenever necessary, a Satisfiability Modulo Theories (SMT) solver, namely Yices [11], for representing and solving the formulas derived from combinatorial problems and tests. An SMT problem is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality. Yices can easily deal with the CITLAB models introduced in Sect. II-A. An SMT instance is a generalization of a boolean SAT instance in which various sets of variables are replaced by predicates from a variety of underlying theories. Obviously, SMT solvers provide a much richer modeling language than that provided by SAT solvers. We have embedded Yices in CITLAB using JNA (Java Native Access) and we exploit the following commands of the Yices APIs (besides those for creating domains and variables):

- `mk_context` creates the logical context.
- `assert` asserts a constraint in the logical context. After one assertion, the logical context may become inconsistent.
- `push` creates a backtracking point. The logical context can be viewed as a stack of contexts.
- `pop` backtracks, i.e., it restores the context from the top of the stack, and pops it off the stack.
- `check` checks if the logical context is satisfiable.
- `del_context` deletes the logical context.

In order to check the satisfiability of a formula in Yices, we add the formula to the logical context by the command `assert` and then we execute the command `check`. Whenever we need to check the satisfiability of a set of formulas having a common subset, we can do, in order to decrease the computation time, an incremental checking by exploiting the commands `push` and `pop`.

We use the SMT solver not only as satisfiability checker, but also as an equivalence prover, thanks to Thm. 7.

C. Desired properties of combinatorial models and tests

In this section we introduce some properties that should be proved in order to assure that a combinatorial model and tests have some quality attributes. These properties refer to attributes that are defined independently from the particular combinatorial specification to be analyzed and they should be true in order to guarantee a certain degree of quality for the combinatorial model and tests. For this reason, we call them *meta-properties*. The violation of a meta-property always means that a quality attribute is not met and may indicate a potential/actual fault in the model or in the tests. Although we will actually define the meta-properties in the following sections, we introduce now three generic categories of quality attributes.

- **Consistency** requires that there are no elements that conflict with each other. For instance, the constraints should not be contradictory (see Sect. III-A).
- **Completeness** requires that every feasible requirement must be covered by at least one test (see Sect. IV-A).
- **Minimality** guarantees that the specification does not contain elements defined or declared in the model but never used. These defects are also known as *over-specification*. For example, if a parameter value is never used, it could be removed from the parameter domain (see Sect. III-C). Another minimality meta-property checks that the test suite is *minimal* (see Sect. IV-B).

III. VALIDATION OF CIT MODELS

A. Inconsistent constraints

In classical deductive propositional logic, a theory is consistent if it does not contain a contradiction. A simple syntactic contradiction happens when the theory contains both the formula φ and its contradiction $\neg\varphi$. In general, we consider a theory (semantically) consistent if and only if it has a model, i.e., there exists an interpretation under which all formulas in the theory are true. In brief, the theory is satisfiable.

We can extend the concept of consistency to combinatorial models with constraints. Given a set of parameters together with their domains, we can check if the constraints over these parameters are consistent, i.e., if they actually allow at least a possible valid assignment to every parameter of the model. An inconsistent set of constraints restricts too much the problem space to the point that no solution is possible.

Algorithm 1 Algorithm for finding a maximum consistent subset

```
Require: an inconsistent set of constraints $C$
Ensure: it returns a consistent subset of $C$
  for $i = (|C| - 1), \ldots, 1$ do
      for all $\{C' \subset C : |C'| = i\}$ do
          if $isConsistent(C')$ then
              return $C'$
          end if
      end for
  end for
  return $\emptyset$
```

Definition 9. A set of constraints $C = \{c_1, \ldots, c_n\}$ is consistent if $\bigwedge_{i=1}^{n} c_i$ is satisfiable. A model is consistent if its constraints are consistent.

Example 10. An example of inconsistent set of constraints is $\{a \wedge \neg b, a \rightarrow b\}$.

In order to discover if a model is consistent, we use the SMT solver Yices by simply checking the satisfiability of the conjunction of the constraints.

How to deal with inconsistent constraints: Once the model has been proved inconsistent, the designer is interested in identifying which constraints are responsible for such inconsistency. In the simplest case a *single* constraint c_i is inconsistent by itself, i.e., it is a contradiction ($\models \neg c_i$). Single inconsistencies must be identified and removed (or corrected).

Example 11. For example, the constraint # a=5 **and** a=6 # is inconsistent.

Identifying single inconsistencies using Yices is easy: it is sufficient to check each constraint for satisfiability.

However, in most cases there is not a single inconsistent constraint, but the inconsistency derives from the interaction of the constraints. In this case, the designer may be interested in finding a maximum subset of consistent constraints.

Definition 12. Given an inconsistent set Γ of constraints, we say that Ω is a *maximum consistent* (or *satisfiable*) subset (MCSS) of Γ, if $\Omega \subset \Gamma$ and every subset Δ (such that $\Omega \subset \Delta \subseteq \Gamma$) is inconsistent.

Finding the MCSS can be done by a greedy algorithm, as the one shown in Alg. 1. It checks for consistency all the proper subsets of C, going from the biggest ones to the singletons. As soon as a consistent subset C' is found, it is returned. The proposed algorithm has the advantage of returning a consistent subset, but does not precisely identify the causes of the original inconsistency: we can only know that the constraints in $C \setminus C'$ are inconsistent with at least one of the constraints in C'. In order to exactly discover the constraints responsible for the inconsistency, one should use an algorithm for finding the *minimum unsatisfiable core* [20], i.e., the smallest set of constraints that is still unsatisfiable.

Example 13. Let's consider the inconsistent set of constraints $C = \{a \wedge \neg b, a \rightarrow b, a \vee b\}$. The algorithm first selects the subsets of size $|C| - 1$; the subset $C_1 = \{a \wedge \neg b, a \rightarrow b\}$ is still inconsistent, subsets $C_2 = \{a \wedge \neg b, a \vee b\}$ and $C_3 = \{a \rightarrow b, a \vee b\}$, instead, are consistent. The algorithm returns the first subset found consistent (C_2 or C_3); note that the two subsets are not equivalent: C_2 admits only $\{a = true, b = false\}$, while C_3 admits two tests different from the one admitted by C_2. The modeler should check if the returned set actually captures the intended requirements of the system.

B. Constraints Vacuity

In this section we extend the notion of vacuity to combinatorial constraints. Generally, vacuity has been applied to properties of behavioral models in formal verification. A property is vacuously satisfied if that property is satisfied and proved true regardless of whether the model really fulfills what the specifier originally had in mind or not. For example, the property $a \rightarrow b$ is vacuously satisfied by any model where a is never true. Vacuity is an indication of a problem in either the model or the property. Several techniques to detect vacuity have been proposed (e.g., [3], [18]) and also tools that perform vacuity detection have been developed (e.g., [13]). In classical formal verification, to detect vacuity as defined in [3], [18], it is enough to replace parts of the property and see if the replacement has any effect on the result of the verification. A common technique to detect vacuity [18] consists in replacing a subformula ϕ of property φ with *true* or *false* (depending on the polarity of ϕ in φ) and checking for its satisfiability.

In our case, however, the constraints do not represent properties that can be derived from the model, but the model itself, i.e., the whole and only formal specification, and the classical definitions do not apply. We still borrow the term *vacuity* to indicate a constraint or one of its subformulas which is useless. Intuitively, a constraint, or a part of it, is vacuous if it can be removed, either because it is always true or because it is implied by the other constraints. We distinguish between total vacuity and partial vacuity.

Definition 14. A constraint c_i is *totally vacuous* iff $\models \left(\bigwedge_{k \in \{1,\ldots,n\}-\{i\}} c_k\right) \rightarrow (c_i \equiv true)$.

Intuitively, a constraint is totally vacuous if it is implied by the other constraints which make the constraint useless, since it does not add any further restriction to the model.

Example 15. Let's consider the set of constraints $C = \{\neg a, a \rightarrow b\}$. The constraint $a \rightarrow b$ is totally vacuous. Indeed, $\neg a \rightarrow (a \rightarrow b)$ is valid.

Tautologies used as constraints are a special case of total vacuity.

Example 16. Let's consider the set of constraints $C = \{a \wedge b, c \vee \neg c\}$. The constraint $c \vee \neg c$ is totally vacuous, since it is a tautology.

A constraint could be partially vacuous, when it contains an occurrence of a subformula ϕ which is useless, i.e., it is implied by the other constraints and by the other part of

Algorithm 2 Given a predicate R, REDUCE returns the set of all the formulas obtained from R by removing one occurrence of a subformula in R

```
function REDUCE($R$)
    if $R$ is atomic then
        return $\emptyset$
    else if $R = A \circ B$ then
        $ra \leftarrow$REDUCE($A$)
        $rb \leftarrow$REDUCE($B$)
        return $ra \circ B \cup rb \circ A \cup \{A, B\}$
    else if $R = \neg A$ then
        return $\neg$REDUCE(A)
    end if
end function
```

where
$\circ$ is $\vee$ or $\wedge$
$\{x_1, \ldots, x_n\} \circ R = \{x_1 \circ R, \ldots, x_n \circ R\}$, $\emptyset \circ R = \emptyset$
$\neg\{x_1, \ldots, x_n\} = \{\neg x_1, \ldots, \neg x_n\}$, $\neg\emptyset = \emptyset$

the same constraint. Therefore ϕ could be removed in that occurrence.

In order to discover if a constraint c_i is partially vacuous, we generate, by the function REDUCE reported in Alg. 2, all the possible formulas that can be obtained from c_i by removing only an occurrence of one of its subformulas[3].

Partial vacuity is checked using the following definition.

Definition 17. A constraint c_i is partially vacuous if there exists $\varphi \in \text{REDUCE}(c_i)$ such that $\models \left(\bigwedge_{k\in\{1,\ldots,n\}-\{i\}} c_k\right) \rightarrow (c_i \equiv \varphi)$.

If formula in Def. 17 is true, then φ is equivalent to c_i (assuming all the other constraints). Using φ instead of c_i would give an equivalent simpler model.

Example 18. Let's consider the set of constraints $C = \{c_1, c_2\}$ with $c_1 = a \wedge b$ and $c_2 = (a \vee b) \wedge d$. $\text{REDUCE}(c_2) = \{a \wedge d, b \wedge d, a \vee b, d\}$. The constraint c_2 is partially vacuous because it is equivalent to d: indeed, $c_1 \rightarrow (c_2 \equiv d)$ is valid. Note that the whole constraint is not vacuous: indeed, $c_1 \rightarrow (c_2 \equiv true)$ is not valid.

In order to check a constraint for total vacuity in Yices, we check the formula introduced in Def. 14 for validity. For checking partial vacuity of constraint c_i, we check, for every formula in $\text{REDUCE}(c_i)$, the validity of the formula introduced in Def. 17. Note that one may stop as soon as she/he finds a φ in $\text{REDUCE}(c_i)$ which makes the formula of Def. 17 true.

How to deal with vacuous constraints: The vacuity of a constraint may actually be caused by an error in it. The user may have mistyped an element or an operator and this may cause the vacuity. Instead, if the constraint is correct, it can be eliminated or simplified in order to make the model simpler (and possibly the test generation faster). When a single constraint is totally vacuous, it can be eliminated without problems; note that, after its removal, the vacuity should be checked again, since some of the other vacuous constraints may have become not vacuous. When a constraint c_i is partially vacuous several times, i.e., there exist φ_1 and φ_2 in $\text{REDUCE}(c_i)$ which are equivalent to c_i, c_i can be substituted by either φ_1 or φ_2 and its vacuity must be checked again.

In some cases, however, the user may be interested in keeping also totally vacuous constraints as further properties of the system. It is common in formal modeling having, stated in the model, properties which are implied by the assumptions or axioms asserted in the model. Indeed, a vacuous constraint represents a property of the system under test which is implied by other constraints. So, it should be classified as **property** and not as constraint. We plan to add to the CITLAB language also a notation for properties, together with the support for proving them.

C. Useless parameter values and useless parameters

A parameter p can contain in its domain some values which are never taken by p.

Definition 19. The value v_k^j of a parameter p_j is *useless* if, due to the constraints, p_j can never assume value v_k^j.

Definition 20. If the parameter p can assume only a value, then the whole parameter is *useless*.

We consider such elements useless, since they can be ignored during the test generation process.

In order to discover if the value $v_k^j \in D_j$ of parameter p_j is useless, we check with Yices if $p_j = v_k^j \wedge \bigwedge_{i=1}^{n} c_i$ is unsatisfiable. Totally, we must check $\sum_{j=1}^{m} |D_j|$ values. Once we know which parameter values are useless, we can also identify the useless parameters. Indeed, given a parameter p, if all its values except one are useless, then p can be classified as useless. Note that, if the model is consistent, each parameter can take at least one value. Uselessness checking should be done only on consistent models.

How to deal with useless elements and parameters: Uselessness of parameters and values can be caused by errors in the constraints: the test designer may have inadvertently introduced a restriction not present in the real system under test. In this case, the constraints should be revised. If this is not the case, the useless parameters and values can be removed from the model. However, if they are contained in some constraints, also the constraints must be modified accordingly.

Parameters removal can ease the test suite generation process, since size reduction of a combinatorial model domain decreases significantly the generation time. If a useless parameter is removed, it may be reintroduced in the tests with its unique value.

Example 21. Consider, for instance, the CITLAB model in Fig. 2. Due to the first constraint (`a == a.a1`), parameter `a` can take only value `a1`. In this case, parameter `a` could be removed during the test generation process and, if necessary, inserted again in the test suite. Due to the second constraint (`b != b.b1`), instead, the domain of `b` can be reduced, excluding value `b1`. In both cases the corresponding constraint must be removed as well.

[3]The algorithm can be easily extended to deal with other boolean operators as $\rightarrow$, $\leftrightarrow$, and $\oplus$.)

```
Model uselessModel
Parameters:
    Enumerative a {a1 a2 a3};
    Enumerative b {b1 b2 b3};
end
Constraints:
    # a == a.a1 #
    # b != b.b1 #
end
```

Figure 2. Example of useless parameter and useless parameter value

IV. Validation of CIT test suites

Test suite validation checks that the test suites produced by a generation tool are correct (see Sect. IV-A) and minimal (see Sect. IV-B).

A. Test Suite Correctness

We introduce the following definitions.

1) A test suite is *sound* if every test is syntactically correct and valid:
   a) an assignment of values to the parameters is a *syntactically correct* test if it satisfies the type definitions;
   b) a test is *valid* if it does not violate any constraint (see Def. 4).
2) A test suite is *complete* if every feasible test requirement is covered.

Definition 22. Test suite correctness A test suite is *correct* if it is *sound* and *complete*.

Checking if a test suite is sound only requires syntax checking and the tests validity assessment. In order to assess if a test t is valid, it is enough to substitute the values of the parameters in t in each constraint c_i and check that every c_i evaluates to true.

Example 23. Let's consider the model with a parameter x defined in the domain $\{1, \ldots, 100\}$ and the constraint $\mathsf{x} > 10$. The following test suites are both not sound:

- $\{\{\mathsf{x} = 101\}\}$: Although the test suite satisfies the constraint, it does not respect the type definition of x (the test is not syntactically correct).
- $\{\{\mathsf{x} = 9\}\}$: The test suite does not satisfy the constraint (the test is not valid).

Checking the completeness of a test suite requires a satisfiability solver, since it is not possible to judge if a test requirement is feasible or not by syntax checking. The completeness check can be performed by Alg. 3. It checks if every tuple is covered by at least one test in TS. If a tuple tp is not covered, it checks its feasibility using the SMT solver and returns false if tp is feasible and not covered.

How to deal with incorrect test suites: An unsound test suite must be fixed before it can be used. There are two main ways: either discard any invalid test or modify it in order to make it valid. Removing an invalid test is easier than fixing it, but it may reduce the testing coverage. On the other hand,

Algorithm 3 Test suite completeness check

```
Require: test suite $TS$ to be checked
Require: the domain $D$ of the parameters
Require: the required $n$-wise coverage
Ensure: it returns true if $TS$ is complete, false otherwise
  $TP \leftarrow$ all the $n$-tuples from $D$
  for all $tp \in TP$ do
    $covered \leftarrow false$
    for all $t \in TS$ do
      if $t$ covers $tp$ then
        $covered \leftarrow true$
        break
      end if
    end for                        ▷ if not covered, check feasibility
    if $\neg covered$ then
      if $tp \wedge \bigwedge_{i=1}^{n} c_i$ is satisfiable then
        return $false$
      end if                       ▷ $tp$ is infeasible
    end if
  end for
  return $true$
```

fixing a test requires a greater effort, since it is not clear in general which assignments in the test are responsible for its invalidity.

An incomplete test suite can still be useful, although it may not exercise the system under test as well as required. The tester may use a test generator tool that accepts an existing possibly incomplete test suite (often called *seeds*) and tries to generate the missed test cases. With a slight modification, Alg. 3 can also be used to measure the incompleteness of a test suite by counting the number of feasible tuples that are not covered. As in classical testing, coverage measures can be used to assess the quality of the test suite and of the tool that has generated it. Our approach would correctly count only the feasible test requirements.

B. Test Suite Minimality

Combinatorial test generators can produce very compact test suites, which, however, could still contain redundant tests. For example, tools generating at every step a test that covers still uncovered tuples, may generate at some point a test which might also cover several other tuples previously covered by tests and these previously generated tests may become useless. An optimum test case should satisfy two objectives simultaneously. First, it must satisfy the maximum number of uncovered requirements (tuples). Second, it must have the minimum overlap in requirements coverage with other test cases. The smallest ideal test suite is that in which each tuple is covered by exactly one test case, but this very seldom can happen: in most cases, a tuple will be covered by many tests, creating possible redundancies. A certain level of redundancy is in general unavoidable. However, some redundancies are useless: some tests that overlap may be eliminated without reducing the total coverage of the test suite. This problem is also known as test suite reduction or minimization [15].

First of all, we want to discover if a test suite could be reduced without losing coverage. We introduce the following

Algorithm 4 Test suite minimality check

```
Require: test suite $TS$ to be checked
Require: set of tuples $TP$
Ensure: it returns true if $TS$ is minimal, false otherwise
  $EssentialTests \leftarrow \emptyset$
  for all $tp \in TP$ do
    $cov \leftarrow \emptyset$
    for all $t \in TS$ do
      if $t$ covers $tp$ then
        $cov \leftarrow cov \cup \{t\}$
      end if
    end for
    if $|cov| = 1$ then
      $EssentialTests \leftarrow EssentialTests \cup cov$
    end if
  end for
  return $|EssentialTests| = |TS|$
```

Algorithm 5 Reduction algorithm

```
Require: test suite $TS$ to reduce
Require: set of tuples $TP$ covered by $TS$
Ensure: it returns a possibly minimal test suite
  $TS' \leftarrow TS$
  $mTS \leftarrow \emptyset$
  while $TP \neq \emptyset$ do
    $t \leftarrow getMostCoveringTest(TS', TP)$
    $CoveredTPs \leftarrow getCoveredTPs(TP, t)$
    if $CoveredTPs = \emptyset$ then
      return $mTS$
    end if
    $mTS \leftarrow mTS \cup \{t\}$
    $TS' \leftarrow TS' \setminus \{t\}$
    $TP \leftarrow TP \setminus CoveredTPs$
  end while
  return $mTS$
```

definition.

Definition 24. A test suite TS is minimal if there exists no subset $TS' \subset TS$ such TS' satisfies all the testing requirements as the original set TS does, i.e., that all the tuples covered by TS are also covered by TS'.

How to recognize a non-minimal test suite? We can postulate that a test t is *redundant* if all the tuples covered by t are also covered by other tests. On the contrary, a test case can be defined *essential*, i.e., it cannot be removed from the test suite, as follows.

Definition 25. A test case t_i is *essential* if it covers at least one tuple tp in TP (the set of all the tuples for a given n-wise coverage) not covered by other test cases of the test suite TS. Formally, $\exists tp \in TP\colon (t_i \models tp \wedge (\neg \exists t_j \in TS\colon (i \neq j \wedge t_j \models tp)))$.

Theorem 26. *A test suite TS is non-minimal iff TS contains at least a not essential test.*

Alg. 4 reports the algorithm we use to check if a test suite is minimal. It checks if each test t of the test suite is essential, i.e., for every tuple *tp* it collects (in cov) the tests that cover *tp*. If cov contains only one test, then that test is essential and collected in *EssentialTests*. If all the tests are essential, then the test suite TS is minimal.

How to deal with non-minimal test suites: Test suite reduction (also known as test suite minimization) is often applied in the context of regression testing, when one wants to find a minimal subset of test cases which satisfy all the testing requirements as the original set does. The problem of finding the minimal test suite that satisfies a set of test goals can be reduced, in polynomial time, to the minimum set covering problem which is NP-hard. A simple greedy heuristic for the minimum set covering problem defined in [7] can be adapted to the test suite minimization.

In this paper, in order to obtain a final test suite with fewer test cases, we try to build a reduced test suite in which the requirements coverage is preserved and all the test cases are essential. Note that, however, if the test suite is non-minimal, then one cannot simply remove all the not essential test cases, since a not essential test case may become essential after another not essential test case is removed from the test suite. The choice of which tests to include in the final test suite is critical. We have implemented a greedy algorithm, reported in Alg. 5, which can reduce the original test suite, still covering all the requirements. At every step it chooses the test that covers most uncovered tuples. When there is a tie between multiple test cases, one test case is randomly selected.

Example 27. Let's consider the test suite $TS = \{t_1, t_2, t_3\}$ whose tests cover, respectively, requirements $\{a, b\}$, $\{a, c\}$ and $\{b, d\}$. If in the first iteration the greedy algorithm collects test t_2, in the second iteration it must collect the test t_3 since it covers most uncovered tuples. At this point, all the tuples have been covered. So, the greedy algorithm reduces TS as $mTS = \{t_2, t_3\}$.

Note that the algorithm may fail to reduce the test suite, even if this test suite is non-minimal.

Example 28. Let's consider the same test suite shown in Example 27. If in the first iteration the greedy algorithm collects test t_1, it must also collect both tests t_2 and t_3 in two following iterations. So the final test suite is not minimized, i.e., $mTS = TS$. However, as seen in Example 27, TS is non-minimal.

V. Experimental results

As model set for CIT problems we have gathered a set of 64 models with constraints taken from the literature (CASA [10], FoCuS [23], ACTS [19], and IPO-S [4]) and used (in subsets) also by many other papers. We have implemented the validation framework in CitLab and we have performed experiments over models and their test suites generated with different tools. The benchmarks are available at the CitLab web site[4]. We have performed the experiments on a Linux PC with an i7 processor 3770 (3.4 GHz) and 16 GB of RAM.

[4]https://code.google.com/a/eclipselabs.org/p/citlab/

Table I
VACUOUS CONSTRAINTS

Model (# constr.)	# useless subform.	Vacuous constraints				
		partially (but not totally)		totally		
		#	%	#	%	Sum %
bench_01 (24)	7	4	16.7	2	8.3	25.0
bench_02 (22)	2	2	9.1	0	0.0	9.1
bench_03 (10)	0	0	0.0	1	10.0	10.0
bench_04 (17)	11	7	41.2	2	11.8	52.9
bench_05 (39)	9	0	0.0	1	2.6	2.6
bench_06 (30)	40	8	26.7	22	73.3	100.0
bench_07 (15)	20	3	20.0	11	73.3	93.3
bench_08 (37)	22	6	16.2	11	29.7	45.9
bench_09 (37)	51	13	35.1	24	64.9	100.0
bench_10 (47)	31	9	19.1	16	34.0	53.2
bench_11 (32)	11	5	15.6	6	18.8	34.4
bench_12 (27)	12	5	18.5	4	14.8	33.3
bench_13 (26)	17	9	34.6	4	15.4	50.0
bench_14 (15)	10	5	33.3	3	20.0	53.3
bench_15 (22)	3	3	13.6	0	0.0	13.6
bench_16 (34)	41	10	29.4	22	64.7	94.1
bench_17 (29)	0	0	0.0	1	3.4	3.4
bench_18 (28)	3	0	0.0	3	10.7	10.7
bench_19 (43)	6	2	4.7	0	0.0	4.7
bench_20 (48)	30	12	25.0	14	29.2	54.2
bench_21 (46)	65	12	26.1	32	69.6	95.7
bench_22 (22	17	6	27.3	10	45.5	72.7
bench_23 (15)	13	4	26.7	7	46.7	73.3
bench_24 (29)	9	3	10.3	6	20.7	31.0
bench_26 (32)	16	3	9.4	11	34.4	43.8
bench_27 (20)	3	1	5.0	3	15.0	20.0
bench_28 (37)	6	1	2.7	3	8.1	10.8
bench_30 (35)	38	10	28.6	17	48.6	77.1
CommProt (128)	814	73	57.0	24	18.8	75.8
Concurr (7)	7	1	14.3	0	0.0	14.3
gcc (40)	6	2	5.0	0	0.0	5.0
HealthC2 (25)	14	2	8.0	0	0.0	8.0
ProcComm2 (125)	43	7	5.6	97	77.6	83.2
Services (388)	81	27	7.0	0	0.0	7.0
SmartHome (43)	33	16	37.2	17	39.5	76.7
Storage1 (95)	205	41	43.2	0	0.0	43.2
Telecom (21)	5	1	4.8	0	0.0	4.8

A. *Consistency of constraints*

The aim of this experiment is to determinate if some of the models under test present any inconsistent constraint. The constraints validation process performed over the 64 models took about 5.6 seconds and it proved that all the models have only consistent constraints. This result was expected since these models have been extensively used for test generation and no test can be generated from inconsistent models.

B. *Vacuity detection*

The aim of this experiment is to determine the presence of partially/totally vacuous constraints. The experiment took 45.5 seconds. Table I reports the 37 models that present at least a form of vacuity; for each model, we report in round brackets the number of its constraints. The second column reports the number of (occurrences of) subformulas that cause a partial vacuity in a constraint: such subformulas are *useless*. The next columns report the number and the percentage of constraints that are partially vacuous, but that are not also totally vacuous. Then, the number and the percentage of totally vacuous constraints are reported. Finally, the table shows the percentage of constraints that have any form of vacuity (partial or total). We can notice that there are more useless subformulas than partially vacuous constraints: indeed, a partially vacuous constraint can have more than one useless subformula. The number of totally vacuous constraints can be high (e.g., 77.6% for *ProcComm2*); however, not all the vacuous constraints can be removed at once. As explained in Section III-B, in order to remove vacuity, one should remove a vacuous constraint at a time, and check for vacuity after each removal: indeed, after the removal of a vacuous constraint, (some of) the other vacuous constraints could become no more vacuous.

The constraints vacuity has proved to be a widespread problem. Almost all the benchmarks *bench_n*, that have been "randomly synthesized starting from real case studies" [10], manifest this problem. However, many real life models are affected too. In general, vacuity is difficult to detect by hand and a tool like that presented in this paper is essential for discovering it.

Table II
USELESS PARAMETER VALUES AND USELESS PARAMETERS

	Useless parameter values			Useless parameters			
	Bool	Enum	Total	Bool	Enum	Total	
						#	%
bench_01	2	0	2	2	0	2	2.06
bench_02	1	0	1	1	0	1	1.06
bench_04	2	1	3	2	0	2	3.45
bench_06	12	0	12	12	0	12	15.58
bench_07	4	0	4	4	0	4	13.33
bench_08	3	1	4	3	0	3	2.52
bench_09	10	4	14	10	0	10	16.39
bench_10	4	2	6	4	1	5	3.40
bench_11	3	0	3	3	0	3	3.13
bench_12	3	0	3	3	0	3	2.04
bench_13	5	1	6	5	0	5	3.76
bench_14	2	1	3	2	0	2	2.17
bench_15	1	0	1	1	0	1	1.72
bench_16	13	0	13	13	0	13	14.94
bench_20	8	1	9	8	0	8	5.06
bench_21	13	2	15	13	0	13	15.29
bench_22	7	0	7	7	0	7	8.86
bench_23	1	2	3	1	0	1	3.70
bench_24	2	0	2	2	0	2	1.68
bench_26	3	0	3	3	0	3	3.16
bench_30	8	0	8	8	0	8	10.13
ProcComm2	0	10	10	0	3	3	12.00
SmartHome	14	0	14	14	0	14	36.84

C. *Useless parameter values and useless parameters*

The aim of this experiment is to determine the presence of useless parameters values and useless parameters. Table II shows the results. Checking useless parameters and values over the 64 models took about 6.2 seconds. We have found that 23 models have at least one useless parameter value and one useless parameter. As expected, we note that each useless boolean value corresponds to a useless parameter. Instead, a useless enumerative value does not necessarily imply that the corresponding parameter is useless.

The presence of so many combinatorial models with useless parameters has surprised us, since they have been extensively used in experiments in the literature. We discovered that most of them (*bench_n*) have been randomly synthesized. Nonetheless, they should be fixed in order to make them

Table III
TEST SUITE VALIDATION

Tool	# test suites		
	Complete	Correct	Minimal
ACTS	64	64	**62**
MEDICI	64	64	64
CASA	64	64	64

more plausible as real models and be considered suitable as benchmarks for testing technique. One model (*ProcComm2*), however, claims to be a "real-life test space instance generated by or for our customers" [23]. In another case (*SmartHome*), useless parameters are present because the model has been automatically obtained from a feature model without applying any optimization [6].

D. Test suite validation

We have performed experiments over the test suite generated in 10 different runs with three different tools: ACTS [19], CASA [10] and MEDICI (an internal tool for test generation we are developing). The test suite used was the same of the previous experiment (64 models) and we chose to perform a pairwise generation. Results are shown in Table III. The three tools produced complete and correct test suites for all the models. MEDICI and CASA always produced minimal test suites, while ACTS (which was the fastest in test generation) produced a non-minimal output for two benchmarks. We have applied 10 times the reduction algorithm presented in Alg. 5 to the test suites obtained in these two cases. In all the cases the reduction algorithm was able to reduce the test suite. In one case, however, the produced test suite was still non-minimal.

The time to check if all the test suites are minimal is 50 seconds, while the reduction of one test suite takes 0.8 seconds.

VI. RELATED WORK

For different programming languages, several tools automatically look for common errors as, for example, FindBugs, PMD and Checkstyle for Java, or Splint for C[5]. These tools look for erroneous code but also for *stylistic conventions* violations that may indicate a possible problem. For example, the pattern *Unwritten field* of FindBugs signals if a field has never been written and always returns its default value: the violation of this pattern could show that the field is not necessary or that it must be updated somewhere.

Also for models, some model review techniques have been proposed.

A model review technique has been developed for the Software Cost Reduction (SCR) method [16], a requirements specification method that uses a tabular notation to define mathematical functions. There are different tables: *condition*, *event*, and *mode transition* tables. Each table describes a *variable* or a *mode* as a function of modes and/or events and/or conditions. The authors defined a *formal requirements model* specifying the properties that any SCR specification must satisfy, and developed a tool, the *consistency checker*, for checking these properties. They identified eight categories of properties: *Proper Syntax*, *Type Correctnesses*, *Completeness of Variable and Mode Class Definitions*, *Initial Values*, *Reachability*, *Disjointness*, *Coverage* and *Lack of Circularity*. Some properties are similar to ours. For example, *Coverage* requires that at least one condition in each row of a condition table must be true; this is similar to our consistency check.

The UML state machines are an object-based variant of Harel statecharts. In [22] the authors present a set of rules that seek to avoid common types of errors by ruling out certain modeling constructs for UML state machines or Statecharts. The authors state that the first rules that must be respected are the UML *well-formedness rules*. These rules are expressed as OCL constraints over UML models; the satisfaction of these constraints assures the syntactical correctness, which is a prerequisite for executing more complex checks. An example of *well-formedness rule* is the rule *CompositeState-1* that states that *a composite state can have at most one initial vertex*. The authors then reviewed different style guides proposed for statecharts and their dialects. They devised two categories of rules. *Syntactical Robustness Rules* identify syntactical constructions that, although syntactically correct according to the *well-formedness rules*, should be avoided because they could produce misleading models. *Semantic Robustness Rules* try to detect incorrect model behaviours, e.g., race conditions. Some syntactical robustness rules map to our checks. For instance, rules *MiracleState* and *Connectivity*, requiring that each state is reachable, are similar to our useless parameter and value check since they all require model minimality.

In [1] a model review technique is proposed for Abstract State Machines (ASMs), an extension of Finite State Machines. Seven meta-properties have been devised for checking the consistency, the completeness and the minimality of ASM models. Some of these meta-properties inspired our current work. For example, a meta-property requires that *every controlled function can take any value in its co-domain*: this meta-property is similar to the control we do to check that every parameter value is useful (see Section III-C).

A model review technique has been developed also for models of the NuSMV model checker [2]; the authors identified ten meta-properties for checking the consistency, the completeness and the minimality of NuSMV models. We have been inspired also by some of these meta-properties. One meta-property checks that the temporal properties of the NuSMV specification are not vacuously satisfied: in a similar way, we check that the constraints are not totally/partially vacuous. Note that, however, our definition of vacuity is slightly different from the classical definitions [3], [18].

Test suite reduction has been extensively used in regression testing. Chavatal [7] proposes the use of a greedy heuristic that selects at a time a test case that covers most yet to-be-covered requirements, until all requirements are satisfied. Our algorithm in Alg. 5 is an instantiation of that proposed by

[5]http://findbugs.sourceforge.net/, http://pmd.sourceforge.net/, http://checkstyle.sourceforge.net/, http://www.splint.org/

Chavatal. Harrold and colleagues [15] propose a similar, but improved heuristic that generates solutions that are always as good or better than the ones computed by Chavatal. Their technique selects a representative set of test cases from a test suite that provides the same coverage as the entire test suite. This selection is performed by identifying, and then eliminating, the redundant and obsolete test cases. We plan to translate Harrold's algorithm also for combinatorial test suites.

VII. Conclusions

We have presented a set of quality checks that CIT models and test suites should pass: the constraints are consistent and do not contain useless parts, all the parameters and values are useful, test suites do not violate the constraints and cover all the testing requirements, and the final test suite does not contain tests that can be removed without loss of coverage. We have devised suitable techniques, based on an SMT solver, to perform these checks and established some policies about how to deal with violations of these properties.

We have identified the following use cases of our validation framework involving different kinds of users. The first set of users are the *clients* of combinatorial testing, i.e., those who apply combinatorial testing to real systems. Our validation framework helps *designers* to identify defects in models: an inconsistent or vacuous constraint and a useless parameter or value are often an evidence of a defect in the model. *Testers* can use our validation framework in order to assess the quality of the test suites they use. The test suite may require to be fixed and some tests may be discarded because they are wrong or useless. The measure of coverage can be used by testers to determine the quality of the testing process.

The second set of users are the *providers* of CIT tools and frameworks. By using our validation framework, researchers can check the quality of the models they use for benchmarking and experimenting their algorithms. Moreover, the validation framework can be used to validate (and eventually debug) new test generation algorithms, to see if the test suites they generate are actually correct and minimal. The presence of our validation component inside the proposed CIT framework CitLab makes also the comparison and integration of test generation techniques more fair, since it guarantees a way to ensure that every tool embedded in CitLab is producing correct results.

In this paper we focus on detecting potential defects and we give generic guidelines for removing them. As future work, we plan to study the impact of defect removal over the test generation process, especially in terms of test suite size and test generation time.

Acknowledgment

This work is partially supported by GenData 2020, a MIUR PRIN 2010-11 project.

References

[1] P. Arcaini, A. Gargantini, and E. Riccobene. Automatic review of Abstract State Machines by meta property verification. In C. Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215*, pages 4–13, Langley Research Center, Hampton VA 23681-2199, USA, April 2010. NASA.

[2] P. Arcaini, A. Gargantini, and E. Riccobene. A model advisor for NuSMV specifications. *Innovations in Systems and Software Engineering*, 7(2):97–107, 2011.

[3] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. In *Proc. 9th International Computer Aided Verification Conference*, number 1254 in Lecture Notes in Computer Science, pages 279–290, 1997.

[4] A. Calvagna and A. Gargantini. T-wise combinatorial interaction test suites construction based on coverage inheritance. *Software Testing, Verification and Reliability*, Special issue on Model Based Testing, 2011. JohnWiley&Sons.

[5] A. Calvagna, A. Gargantini, and P. Vavassori. Combinatorial interaction testing with CitLab. In *Sixth IEEE International Conference on Software Testing, Verification and Validation - Testing Tools Track.*, 2013.

[6] A. Calvagna, A. Gargantini, and P. Vavassori. Combinatorial testing for feature models using citlab. In *The 2nd International Workshop on Combinatorial Testing (IWCT 2013) In conjunction with ICST 2013, March 18-22, Luxembourg*, ICSTW '13, pages 338–347, Washington, DC, USA, 2013. IEEE Computer Society.

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

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

[9] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *ISSTA International symposium on Software testing and analysis*, pages 129–139, New York, NY, USA, 2007. ACM Press.

[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 Transactions on Software Engineering*, 34(5):633 –650, sept.-oct. 2008.

[11] B. Dutertre and L. de Moura. The Yices SMT solver. Technical report, SRI Available at http://yices.csl.sri.com/tool-paper.pdf, 2006.

[12] A. Gargantini and P. Vavassori. CitLab: a Laboratory for Combinatorial Interaction Testing. In *Workshop on Combinatorial Testing (CT) - ICST*, pages 559–568, Montreal, Canada, 2012. IEEE Computer Society.

[13] M. Gheorghiu and A. Gurfinkel. VaqUoT: A tool for vacuity detection. In *Posters & Research Tools Track, FM 2006*, 2006.

[14] M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies: a survey. *Softw. Test, Verif. Reliab*, 15(3):167–199, 2005.

[15] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(3):270–285, 1993.

[16] C. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.

[17] R. Kuhn, R. Kacker, Y. Lei, and J. Hunter. Combinatorial software testing. *Computer*, 42(8):94 –96, aug. 2009.

[18] O. Kupferman and M. Y. Vardi. Vacuity detection in temporal model checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(2):224–233, 2003.

[19] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing. *Software Testing, Verification and Reliability*, 18(3):125–148, Sept. 2008.

[20] I. Lynce and J. P. Marques-Silva. On computing minimum unsatisfiable cores. In *Online Proceedings of SAT 2004, 10-13 May 2004, Vancouver, BC, Canada*, 2004.

[21] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Comput. Surv*, 43(2):11, 2011.

[22] S. Prochnow, G. Schaefer, K. Bell, and R. von Hanxleden. Analyzing robustness of UML State Machines. In *MARTES 06*, pages 61–80, 2006.

[23] I. Segall, R. Tzoref-Brill, and E. Farchi. Using binary decision diagrams for combinatorial test design. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 254–264, New York, NY, USA, 2011. ACM.