Using Binary Decision Diagrams for Combinatorial Test Design

Itai Segall
IBM, Haifa Research Lab
Haifa University Campus
Haifa, 31905, Israel
itais@il.ibm.com

Rachel Tzoref-Brill
IBM, Haifa Research Lab
Haifa University Campus
Haifa, 31905, Israel
rachelt@il.ibm.com

Eitan Farchi
IBM, Haifa Research Lab
Haifa University Campus
Haifa, 31905, Israel
farchi@il.ibm.com

ABSTRACT

Combinatorial test design (CTD) is an effective test planning technique that reveals faulty feature interaction in a given system. The test space is modeled by a set of parameters, their respective values, and restrictions on the value combinations. A subset of the test space is then automatically constructed so that it covers all valid value combinations of every t parameters, where t is a user input. Various combinatorial testing tools exist, implementing different approaches to finding a set of tests that satisfies t-wise coverage. However, little consideration has been given to the process of defining the test space for CTD, which is usually a manual, labor-intensive, and error-prone effort. Potential errors include missing parameters and their values, wrong identification of parameters and of valid value combinations, and errors in the definition of restrictions that cause them not to capture the intended combinations. From our experience, lack of support for the test space definition process is one of the main obstacles in applying CTD to a wide range of testing domains.

In this work, we present a Cartesian product based methodology and technology that assist in defining a complete and consistent test space for CTD. We then show how using binary decision diagrams (BDDs) to represent and build the test space dramatically increases the scalability of our approach, making it applicable to large and complex real-life test design tasks, for which explicit representation of the test space is infeasible.

Finally, we show how BDDs can be used also to solve the CTD problem itself. We present a new and highly effective BDD-based approach for solving CTD, which finds a set of tests that satisfies t-wise coverage by subset selection. Our approach supports also advanced requirements such as requirements on the distribution of values in the selected tests. We apply our algorithm to real-life testing problems of varying complexity, and show its superior performance.

ISSTA'11, July 17–21, 2011, Toronto, ON, Canada

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Verification, Reliability

Keywords

Combinatorial Test Design, CTD, Binary Decision Diagrams, BDDs, Test Planning, Cartesian Products

1. INTRODUCTION

As software systems become increasingly more complex, verifying their correctness is even more challenging. The introduction of service-oriented architectures (SOA) [13] contributes to the growing trend of highly configurable systems, in which many optional features coexist and might unintentionally interact with each other in a faulty way. Verification approaches such as formal verification and model based testing are highly sensitive to the size and complexity of the software, and might require extremely expensive resources. Functional testing, on the other hand, is prone to omissions, as it always involves a selection of what to test from a possibly enormous space of scenarios, configurations, or conditions. It therefore requires careful consideration of what to include in the testing. The process of test planning refers to the definition and selection of tests out of a test space, with the goal of eliminating redundancy and reducing the risk of bugs escaping to the field as much as possible. Combinatorial test design (CTD), is an effective test planning technique, in which the space to be tested is modeled by a set of parameters, their respective values, and restrictions on the value combinations. A subset of the space is then automatically constructed so that it covers all valid value combinations (a.k.a interaction) of every t parameters, where t is a user input. The most common application of CTD is known as pairwise testing, in which the interaction of every pair of parameters must be covered.

The reasoning behind CTD is the observation that in most cases the appearance of a bug depends on the combination of a small number of parameter values of the system under test. Experiments show that a test set that covers all possible pairs of parameter values can typically detect 50% to 75% of the bugs in a program [28, 11]. Other experimental work has shown that typically 100% of bugs can be revealed by covering the interaction of between 4 and 6 parameters [19].

Many algorithms and tools exist for solving the CTD problem, i.e., finding a set of tests that satisfies t-wise cover-

age [15, 24]. However, little consideration has been given to the process of defining the test space to be given as input to the CTD algorithm. This process is incremental and labor-intensive, and involves finding a set of parameters and values that define the test space, and correctly identifying what are the valid (or invalid) value combinations. One of the biggest pitfalls of this process is omissions, i.e., failing to include an important parameter, value or combination of parameter values in the test space. Another pitfall is failing to correctly define the restrictions so that they capture the intended combinations. The lack of tool support for defining the test space limits the use of CTD in practice to simple cases, such as lists of configurations in which the parameters as well as the restrictions on value combinations are obvious.

Binary decision diagrams (BDDs) [3] are a compact data structure for representing and manipulating Boolean functions. In this paper we show how BDDs can be effectively used both for the process of defining the test space and for solving the CTD problem.

First, we present a methodology that assists the tester in defining the test space. This methodology is based on our extensive experience of working in close collaboration with development and testing groups, aiming at avoiding bugs that are the result of test omissions. The methodology consists of listing the Cartesian product of the parameter values, and incrementally excluding invalid combinations. Projection of the Cartesian product on subsets of parameters and filtering of the Cartesian product according to parameter values are essential steps in making the exclusion process effective. We then show how we can use BDDs to represent the Cartesian product, project and filter it. Using BDDs dramatically increases the size of the test space that can be handled, compared to an explicit representation.

Next, we describe how BDDs can be used to solve the CTD problem. Existing algorithms assume that the test space is too huge to be represented explicitly, and use various techniques to try and construct an optimized test plan without explicitly enumerating all possible value combinations. The key idea behind our BDD-based approach is to exploit the compactness of BDDs in order to symbolically hold the entire test space, partitioned according to different criteria. The set of tests that satisfies t-wise coverage is constructed by subset selection, where selection of tests is made from each partition, until the requested interaction level is met. The use of BDDs also enables effectively supporting requirements on the values distribution in the selected set of tests.

The Cartesian product based methodology and the BDD-based CTD are implemented in our tool, FoCuS [14], and have been used to define and solve complex and large testing problems.

In the following we present in Section 2 background on CTD and BDDs as well as related work. Section 3 presents our methodology for defining the test space, and how it is implemented using BDDs. In Section 4 we present our BDD-based algorithm for CTD. Section 5 presents promising experimental results both on standard benchmarks and on real-life test design tasks. Finally, Section 6 draws our conclusions and future research directions.

2. BACKGROUND

2.1 Combinatorial Test Design

Combinatorial test design (CTD) is a well-known, powerful technique for functional testing, in which all value combinations of size t of the parameters of a system are tested. Traditionally, the problem of finding a minimal set of tests that covers all combinations of size t is defined by the mathematically equivalent problem of finding a covering array of strength t [17]. A covering array, $CA(N, t, k, v)$, is an $N \times k$ array on v symbols with the property that every $N \times t$ subarray contains all ordered subsets of size t from the v symbols at least once.

In this paper, we consider the CTD problem as a subset selection problem, in line with our approach for solving CTD, which is described in Section 4. More formally, we define the CTD subset selection problem as follows: Let $P = \{p_1, \ldots, p_n\}$ be an ordered set of parameters, $V = \{V_1, \ldots, V_n\}$ an ordered set of value sets, where V_i is the set of values for p_i, C a set of Boolean constraints over P, and t an interaction level. A test $(v_1, ...v_n)$, where $\forall_i,\ v_i \in V_i$, is a tuple of assignments to the parameters in P. A valid test is a test that satisfies all constraints in C. The set of all valid tests is denoted by $S(P, V, C)$. A subset of the valid tests $S' \subseteq S$ covers all interactions of size t, if every assignment of values to every subset of P of size t that occurs in some test in S, occurs in some test in S'. S' is minimal iff for all $S'' \subseteq S$ that covers all interactions of size t, $|S'| \leq |S''|$.

Many algorithms and tools exist for solving the CTD problem. In [15], Grindal et al. count more than 40 papers and 10 strategies. The pairwise testing website [24] lists over 30 tools that support pairwise testing. In [5], Cohen et al. classify the algorithms for constructing a solution for CTD according to three different approaches: mathematical construction, greedy algorithms, and meta-heuristic search. Construction of a covering array by mathematical techniques such as in [17, 16] can be efficient and usually results in an optimal solution, however these techniques are not applicable to the general case. Since the general problem of finding a minimal set of test cases that satisfies t-wise coverage can be NP-complete [29], most algorithms, including ours, resort to greedy, heuristic-based incremental construction of the test set. This greedy approach usually results in a sub-optimal solution. Finally, the meta-heuristic search approach refers to search-based algorithms such as simulated annealing, genetic algorithms and tabu search [6, 23] that converge to a near-optimal solution after a certain number of iterations.

In [9], Czerwonka describes advanced features of CTD beyond the basic test set construction, that increase the usability of CTD and are essential to make it practically applicable. These features include, among others, mixed-strength generation, which refers to the ability to require different orders of combinations for different subsets of parameters, testing with negative values that indicate error conditions, specifying expected results, and assigning weights to values. These features are all supported by our tool FoCuS and from our experience are indeed required in practice. Specifically, assigning weights to values is a requirement on the distribution of values in the solution test set. This requirement reflects the importance of different values to the tester. For example, it can reflect available resources for testing, such as the ratio between different operating systems installed on

available machines. Since minimizing the size of the test set is the primary concern, the distribution requirement is treated as a soft constraint. However, when the distribution of values in a constructed test set is far from the requested one, the test set might not be usable. Thus, in practice, there is a trade-off between the two concerns. In Section 5, we present results that efficiently support the distribution requirement.

Existing work on facilitating the CTD test space definition mostly concentrates on the ability to state complex restrictions, preferably as propositional formulas or predicates [9, 11, 4]. [5] surveys restriction handling techniques in existing tools, and recognizes the importance of the ability to support full restrictions, i.e., without remodeling of the parameters or explicit enumeration of all invalid tests. While FoCuS supports full restrictions, stated as Boolean expressions in Java syntax, in this paper we concentrate on a different aspect of the CTD test space definition - the need to review the currently constructed test space, mainly to identify new restrictions and parameters, and to reveal omissions and other errors in the specification. The concept of listing the Cartesian product of parameters and excluding invalid combinations exists also in the Spec Explorer model-based testing tool [26]. However, it requires writing model programs that model the behavior of the system under tests. The Cartesian product can be constructed for the parameters of a method of the program, and as such is of limited size. In addition, it does not support operations on the Cartesian product such as projection, expansion, filtering, and viewing invalid combinations.

2.2 Binary Decision Diagrams

Binary decision diagrams (BDDs) [3] are a compact data structure for representing and manipulating Boolean functions, traditionally used in formal verification of hardware and software [12] and in logic synthesis [21][1]. A BDD is a rooted, directed, acyclic graph, in which non-terminal nodes represent Boolean variables, outgoing edges from non-terminal nodes represent values for these variables, and terminal nodes represent Boolean decisions. Given values for the Boolean inputs, the result of the function is achieved by traversing the graph accordingly until a terminal node is reached.

For example, consider the BDD in Figure 1, that represents the function $f(a,b) = a \vee b$: it has two non-terminal nodes, one for a and one for b (note that in general, a variable may appear in more than one non-terminal node). An evaluation of f, given an input (a,b), starts from the root, and considers the value of a first. If it is 1, the result of 1 is reached immediately without considering the value of b. Alternatively, if $a = 0$, the value of b is considered and the result is achieved accordingly.

An important property of BDDs is that of canonicity: given a function, and an order for the variables, there is only one BDD that represents this function. This is achieved by applying two reduction rules – merging of isomorphic subgraphs, and elimination of nodes with isomorphic children.

Due to the canonic representation, comparison of functions represented by BDDs may be performed in constant time. In addition, standard operations on Boolean functions, such as conjunction, disjunction and negation can also be computed efficiently – negation in constant time, conjunction and disjunction in the worst case in time proportional to the product of the input BDD sizes (polynomial time for a constant number of repetitions of the operation), and in practice usually closer to the sum of the input BDD sizes. The "exists" and "forall" quantifiers (for example, $g(x) = \exists y f(x,y)$) are also easy to compute using BDDs. Finally, counting the number of satisfying assignments and iterating over them are other important examples of efficient operations – counting satisfying assignments in time polynomial in the size of the BDD, and iteration in constant time (that is, each step of the iteration is computed in constant time).

[1]We use the most common variant of BDDs, ROBDDS – reduced ordered binary decision diagrams, and use the term BDD for simplicity.

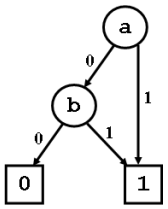

Figure 1: Example BDD, representing the function $f(a,b) = a \vee b$

While formally BDDs represent Boolean functions, they are usually used to represent sets of Boolean vectors. This is achieved by considering characteristic functions. Given a set $F = \{(x_1^1, \cdots, x_n^1), \ldots, (x_1^k, \cdots, x_n^k)\}$, its characteristic function is given by $f(x_1, \ldots, x_n) = 1 \leftrightarrow (x_1, \ldots, x_n) \in F$. Thus, a set of Boolean vectors, F, can be represented symbolically using BDDs by representing its characteristic function. Conjunction, disjunction and negation of these functions correspond to intersection, union and complementation of the sets. Counting satisfying assignments corresponds to computing the size of the set.

Multi-valued decision diagrams (MDDs) [27] are a generalization of BDDs, in which the inputs of the function may be multi-valued. MDDs can be represented using BDDs by encoding multi-valued inputs into binary-coded ones, and MDD operations can be translated into BDD operations [22]. Thus, as a set of Boolean vectors may be represented using BDDs, so can a set of discrete value vectors.

3. DEFINING THE TEST SPACE

3.1 The Methodology

From our experience, one of the main obstacles in applying CTD in real-life scenarios is the construction of the test space. Identifying a complete set of parameters and their values can often be a laborious, error-prone task. In addition, while restrictions are an essential ingredient in almost all real-life test planning efforts, identifying the correct restrictions is often the most complex part of defining correctly the test space. While many tools support restrictions as part of the input, support for identifying these restrictions is usually overlooked.

Our methodology, as supported by the FoCuS tool [14], aids the tester in constructing the test space efficiently, while considerably reducing the risk of omissions. The methodology relies on the concept of Cartesian products. The user may examine the Cartesian product of any set of parame-

ters and values, while considering the validity of each entry in this Cartesian product. The user may choose a set of parameters and values that induce a small Cartesian product, that can be used to explicitly and thoroughly review a certain aspect of the test space, or a very large one, that may be used for getting a general impression of the currently defined test space. Moreover, the user may interactively add restrictions directly from the Cartesian product GUI.

Consider a small example test space, representing a repair operation of a configuration table. Our example system contains two configuration tables. When a table becomes corrupt, the content of the second table is copied to override it. The repair should fail if there are other ongoing operations. Six parameters are used to represent the test space, as follows.

- `configTable1` and `configTable2` – parameters with values OK and Corrupted, that represent the state of the tables.
- `dynamicIOOngoing`, `saveTableOngoing` and `anotherRepairOngoing` – Boolean parameters that represent different operations that may be ongoing when attempting repair.
- `repairSucceeded` – a Boolean parameter, representing whether the repair operation succeeded.

Figure 2 shows the initial Cartesian product view for the example. Currently no restrictions have been defined, thus all combinations are valid. By inspecting this list, the user may notice some combinations that should be invalid. For example, combinations in which there is an ongoing operation but the repair succeeds. This leads the user to realize that a restriction is missing – one that excludes all combinations in which one of the operations is ongoing but the repair succeeds. There are two ways for the user to add such a restriction:

1. Manually – in FoCuS, restrictions may be expressed as Boolean expressions in Java syntax. In our example, the user may add a restriction that disallows `repairSucceeded && (dynamicIOOngoing || saveTableOngoing || anotherRepairOngoing)`.
2. Via the GUI – restrictions may also be added directly from the Cartesian product view. The user may choose the combinations that should be invalid, and press exclude. Restrictions that disallow the selected combinations will be added automatically

The Cartesian product view is very effective in helping the user to avoid omissions. The user is faced with the Cartesian product of a set of parameters, thus reducing the risk of value combinations being left unnoticed. For example, the full Cartesian product in Figure 2 contains the combination in which neither table is corrupt (parameters `configTable1` and `configTable2` both get the value "OK"). This combination, where neither of the tables is corrupt, might otherwise have been overlooked by the user, since the test space he is building represents table recovery scenarios. The user may decide that he is not interested in testing this scenario. Alternatively, he can choose to include this scenario in the test plan, in which case the expected result of the repair operation must be determined. In our example, we choose the former alternative, and therefore exclude this combination

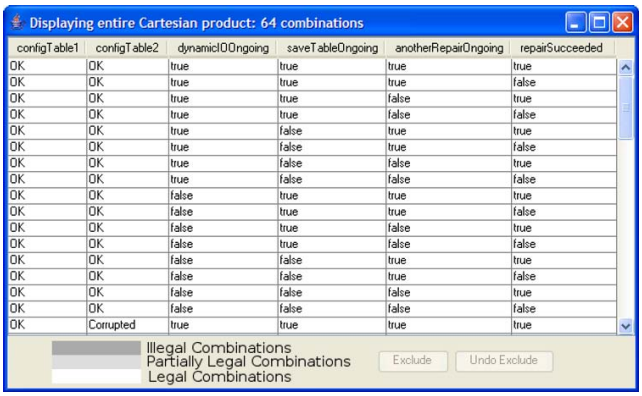

Figure 2: Screenshot of FoCuS showing the entire Cartesian product for the table repair example

from the test space. As before, this can be achieved either by manually adding a restriction, or marking rows in the Cartesian product view and excluding them.

Continuing the example, assume now that the user is interested in the effect of the states of the different configuration tables on the success of the repair operation. Thus, he may ask for a *projection* of the Cartesian product on the parameters `configTable1`, `configTable2` and `repairSucceeded`. Combinations in a projected Cartesian product may be of one of three types:

1. *Valid.* For such combinations, any extension of them to an assignment to the entire set of parameters is valid.
2. *Invalid.* For such combinations, any extension of them to an assignment to the entire set of parameters is invalid.
3. *Partially valid.* These are combinations that are neither valid nor invalid, i.e., a partially valid combination has an extension to an assignment to the entire set of parameters that is valid, and also such an extension that is invalid.

Figure 3 shows the projection of the test space as defined so far on the parameters `configTable1`, `configTable2` and `repairSucceeded`. For example, the combination in which the first table is OK, the second is corrupt and the repair does not succeed is currently valid, meaning that according to the restrictions defined so far, whenever the first table is OK, and the second is corrupt, the repair may fail regardless of the ongoing operations (this should clearly be refined using more restrictions). The two combinations in which both tables are OK are both invalid, meaning that there are no tests in the test space in which both tables are OK, regardless of the ongoing operations (this is due to the restriction introduced above). Finally, the combination in which the first table is OK, the second is corrupt and the repair succeeds is partially valid. This means that there is a combination of ongoing operations for which such a test is valid (specifically, the one in which there is no ongoing operation), and another combination for which such a test is invalid (when there is at least one ongoing operation).

In order to further explore a partially valid combination, upon choosing such a combination, the user may choose to expand this specific combination into more parameters. A

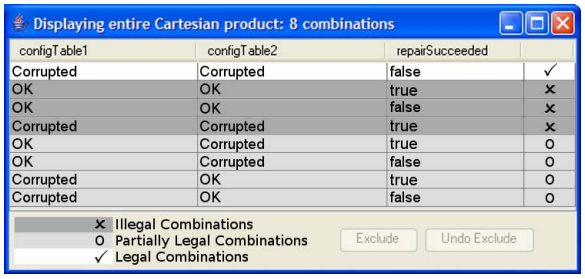

Figure 3: Screenshot of FoCuS showing a projection for the table repair example

new Cartesian product view is produced, in which the current parameters get the values according to the chosen combination, and the newly chosen parameters may get any value. As before, combinations in this Cartesian product may be valid, invalid or partially valid.

The last combination in Figure 3, in which both tables are corrupt and the repair succeeds, is partially valid. The user may choose that this combination should be excluded (thus become invalid). Note that since a projection is now considered, excluding this row from the view actually excludes many different tests from the test space. In this example, it excludes all tests in which both tables are corrupt and the repair succeeds.

Clearly, Cartesian products of real-life testing problems may be very large. In FoCuS, whenever a Cartesian product exceeds some predefined size, it is shown in chunks, i.e., the user sees at any point in time just a small subset of the entire Cartesian product, and may navigate between different chunks. Moreover, the user may also filter the Cartesian product to include only certain values to the parameters, thus reducing the number of combinations viewed at a time.

The methodology, as described so far, is aimed mainly at identifying and adding restrictions. However, it is useful also for identifying missing parameters or values thereof. For example, when reviewing a Cartesian product, a user might notice that a certain combination of values actually stands for two different scenarios, thus realize that another parameter should be introduced in order to distinguish between them.

In Section 5 we use twenty real-life test space instances created by or for our customers. The instances are of varying sizes and complexities, with 5-35 parameters and 0-388 restrictions. The test spaces represent testing problems from different domains, such as telecommunication, healthcare, storage and banking, and for testing different aspects of the system, such as data manipulation and integrity, protocol validation and user interface testing. These were all created in FoCuS using the methodology described here.

3.2 Implementing with BDDs

3.2.1 Test Space Representation

As explained above, a test space is represented by a set of parameters, values for each parameter, and restrictions. The set of valid tests is the set of assignments of values to parameters that are not excluded by restrictions. The tool has to keep track of this set, and of the complementary set of invalid combinations. The naïve way of representing these sets is explicitly – by keeping the explicit list of all the tests that are valid. In order to construct this representation, one has to iterate over all possible assignments to the parameters, and for each assignment compute whether it is allowed by the restrictions. Alternatively, we propose here to use a symbolic representation using BDDs. Clearly, and as demonstrated by the experimental results in Section 5, a symbolic representation using BDDs can support significantly larger test spaces than the naïve explicit implementation.

In our representation, each parameter is represented by one or more binary variables in the BDD. This representation is standard practice in reducing multi-valued decision diagrams into binary ones [22]. In order to build the BDD of valid tests, we first build for each restriction the BDD representing the set of tests allowed by it. Since restrictions in FoCuS are given as Boolean expressions in Java syntax, this is done using a Java parser. A test is valid if and only if it is valid according to all restrictions, therefore the set of valid tests is exactly the intersection of the sets of tests allowed by the restrictions. This is computed by conjuncting the BDDs representing these sets.

For restriction $Rest_i$, we denote by $Allowed_i$ the BDD representing the set of tests allowed by it. We denote by $Valid$ and $Invalid$ the BDDs representing the set of valid and invalid tests in the test space, respectively. Thus, they are computed by the formulas $Valid = \bigwedge_i Allowed_i$ and $Invalid = \neg Valid$, respectively.

Given the BDDs $Valid$ and $Invalid$, one can already answer efficiently questions that are otherwise computationally hard. For example, to find out whether a given test, t, is valid or not, one merely needs to check whether $t \wedge Valid$ is equal to False (it is valid iff the conjunction is not False). The number of valid tests in the test space is exactly the number of satisfying assignments to $Valid$ (which is also easy to compute).

3.2.2 Cartesian Product Views

As mentioned above, the Cartesian product view is the most basic view in FoCuS that aids review and construction of the test space. In this report, the user chooses the parameters of interest, and the tool presents the Cartesian product of these parameters, along with the validity of each entry in this Cartesian product.

When all parameters are chosen, an entry in the Cartesian product is valid (i.e., corresponds to a test in the test space) if and only if it is allowed by all restrictions. The user may choose to view the entire Cartesian product, only the valid tests, or only the invalid ones. The tool presents the corresponding sets and highlights the valid vs. invalid combinations. Given the BDD representation of the test space, the report is easily constructed by iterating over all satisfying assignments to the relevant BDDs ($Valid$ and/or $Invalid$).

When only some of the parameters are chosen for the view, it corresponds to a projection of the test space. As explained above, an entry in this projection may be valid, invalid or partially valid, depending on the validity of its extensions to the parameters that were not chosen.

A projection of a test space may be easily computed using existential and universal quantifiers on BDDs. Given the $Valid$ and $Invalid$ BDDs, and a set of parameters $A = \{a_1, \ldots, a_k\} \subseteq P$ (where P is the set of all parameters), let $B = P \setminus A = \{b_1, \ldots, b_m\}$. The sets of valid, partially valid, and invalid combinations in the projection of the test space

to A are represented by the following BDDs:

1. $Valid_A = \forall b_1 \forall b_2 \cdots \forall b_m \; Valid$
2. $Invalid_A = \forall b_1 \forall b_2 \cdots \forall b_m \; Invalid$
3. $PartiallyValid_A = (\neg \; valid_A) \wedge (\neg \; invalid_A)$

The experimental results in Section 5 show that these are computationally easy operations, even for very large test spaces.

As before, once these BDDs are computed, the set of valid, invalid and partially valid combinations in a projection may be easily shown to the user by iterating over the satisfying assignments to each of these BDDs (where only the variables corresponding to the parameters in the projection are considered).

Two more operations are mentioned in Section 3.1 above – expanding a combination from a projection to more parameters, and filtering out some values for given parameters from the view. The former is implemented by computing another projection of the test space on the newly selected set of parameters, and conjuncting the BDDs of this projection with the selected values for the previously chosen parameters. The latter, i.e., filtering, is also implemented using conjunction – the projection BDDs are conjuncted with a BDD in which all parameters may be equal only to those values that are not filtered out.

4. BDD-BASED COMBINATORIAL TEST DESIGN

We introduce a novel algorithm for Combinatorial Test Design using BDDs. The algorithm is iterative, where in each step it chooses a single test that covers as many new tuples of parameters as possible. This is achieved as follows: in each step the set of test candidates is initialized to all valid tests and is then gradually reduced to contain only tests that cover as many new tuples of parameters as possible, until a single test is chosen from it. BDDs are used in order to keep track of covered and uncovered tuples, and BDD operations are used in order to use these sets for reducing the set of test candidates and eventually select a test that covers many new tuples. The algorithm relies on the efficiency of conjunction and counting satisfying assignments of BDDs, and is given in Algorithm 1.

The algorithm considers a set, T, of the tuples of parameters for which coverage is required. For example, for pairwise coverage, the set T contains all pairs of parameters in the test space. The algorithm maintains for each tuple t in T a BDD $uncov(t)$. This BDD captures the assignments to t (i.e., the combinations of values to parameters in t) that are not covered by the tests that were selected so far. For each tuple t, the BDD $uncov(t)$ is initialized to the projection of the BDD of valid tests on the set of parameters t, denoted by $Proj_t(Valid)$ (Line 1).

In every iteration, the algorithm maintains a BDD $Collected$, which represents the set of tests from which it will choose one at the end of the iteration. $Collected$ is initialized to all valid tests (Line 3). The algorithm iterates over all tuples t in T, and conjuncts the set collected so far with their $uncov$ BDDs (Line 8). Tuples for which the conjunction results in the FALSE BDD are skipped (Line 7). Once iterating over T is complete, the algorithm chooses a random assignment that satisfies $Collected$ (Line 22).

```
   input : Coverage requirements, given as a set T
           of tuples of parameters to cover.
           The BDD Valid of all valid tests.
 1 Init: for t ∈ T do uncov(t) ← Proj_t(Valid)
 2 while T ≠ ∅ do
 3 |  Collected ← Valid
 4 |  Sort T in decreasing order of sat(uncov(t))
 5 |  interrupted ← FALSE
 6 |  for t ∈ T do
 7 |  |  if (Collected ∧ uncov(t)) ≠ FALSE then
 8 |  |  |  Collected ← Collected ∧ uncov(t)
 9 |  |  end
10 |  |  if numNodes(Collected) > sizeThrsld
   |  |  then
11 |  |  |  interrupted ← TRUE
12 |  |  |  Break
13 |  |  end
14 |  end
15 |  if interrupted then
16 |  |  for i = 1 to n do
17 |  |  |  candt_i ←randSat(Collected)
18 |  |  |  newCov_i ←newlyCovered(candidate_i, T)
   |  |  |
19 |  |  end
20 |  |  chosen ← {candt_i | ∀j.newCov_i ≥
   |  |  newCcov_j}
21 |  else
22 |  |  chosen ←randSat(Collected)
23 |  end
24 |  appendResult(chosen)
25 |  for t ∈ T do
26 |  |  uncov(t) ← uncov(t) ∧ ¬chosen
27 |  |  if uncov(t) = FALSE then
28 |  |  |  T ← T \ t
29 |  |  end
30 |  end
31 end
```

Algorithm 1: BDD-Based CTD

The rational behind this iteration is that for a tuple t, the BDD $uncov(t)$ represents the set of all the assignments to all parameters in t that cover some new combination of t. By starting with the set of valid tests, and incrementally conjuncting each $uncov(t)$ with the result of the previous conjunction, we gradually reduce the set of valid tests, to those tests that cover the most new combinations for different tuples. The result of the conjunction is FALSE when there is no valid test that covers new assignments for all the tuples selected so far and for the current one, therefore this tuple is skipped. If the algorithm considers all *uncov* BDDs for conjunction, then the number of newly covered assignments to tuples is exactly the number of BDDs successfully conjuncted (i.e., that were not skipped due to a FALSE result). Thus, in order for the algorithm to cover as many new assignments as possible in each iteration, it should succeed in conjuncting as many *uncov* BDDs as possible. Therefore, in each iteration, the tuples in T are sorted in a decreasing order of the number of satisfying assignments to their *uncov* BDDs (Line 4). This way, BDDs with many satisfying assignments are considered first, leaving as many degrees of freedom as possible for the following tuples.

For large test spaces, the algorithm as described so far might require large amounts of memory or computation time for conjuncting all *uncov* BDDs. Thus, we limit the size of the BDDs used in each iteration (Line 10). If the intermediate BDD exceeds a given size threshold, the iteration on the tuples in T is interrupted, and the intermediate result achieved so far is used (Line 11). In such cases, since not all tuples were explicitly considered, different assignments to the conjunction BDD might cover a different number of new assignments to tuples. Thus, we choose randomly several different assignments to it, and use the one that covers the most new assignments (Lines 16 – 19).

The pseudo-code in Algorithm 1 uses methods `randSat`, `newlyCovered` and `appendResult` to choose a random satisfying assignment satisfying a BDD, count how many new assignments to tuples are covered by a certain test, and add a test to the result of the algorithm, respectively.

4.1 Supporting Weights

As mentioned in Section 2, an advanced feature in CTD, that is important in order to make it useful in practice, is that of weights. Using weights, the user may ask for a certain distribution between the values of a given parameter. Algorithm 2 lists the CTD algorithm, adapted to support weights. The modifications from Algorithm 1 are underlined.

The essence of the modification is in Lines 29 and 34 – whenever a random satisfying assignment is chosen, it is chosen randomly according to a certain distribution, and not necessarily uniformly as before. This distribution is derived initially from the required weights (Line 2), and updated throughout the computation (Line 37). The weights are updated by estimating the overall size of the result, deriving from that the number of times each value is expected to appear in order to match the weight requirements, and subtracting the number of appearances so far for each value.

Another important optimization is given in Lines 4 – 15. Occasionally, when updating the weights, the algorithm concludes that a certain value has appeared enough, and updates its weight to zero. The algorithm tries to use only non-zero weighted values by starting the iteration with the

input : Coverage requirements, given as a set T of tuples of parameters to cover.
The BDD $Valid$ of all valid tests.
Weight requirements, given as a mapping $\underline{Weights : Parameter \rightarrow Value \rightarrow weight.}$

1 Init: **for** $t \in T$ **do** $uncov(t) \leftarrow Proj_t(Valid)$
2 Init: $\underline{curWgts \leftarrow Weights}$
3 **while** $T \neq \emptyset$ **do**
4 $\quad \underline{ValidNonZero \leftarrow Valid}$
5 $\quad$ **for** $\underline{(Attr, Val) \in curWgts}$ **do**
6 $\quad\quad$ **if** $\underline{curWgts(Attr, Val) = 0}$ **then**
7 $\quad\quad\quad \underline{ValidNonZero \leftarrow}$
8 $\quad\quad\quad \underline{ValidNonZero \wedge \neg(Attr = Val)}$
9 $\quad\quad$ **end**
10 $\quad$ **end**
11 $\quad$ **if** $\underline{\exists t \in T : ValidNonZero \wedge uncov(t) \neq FALSE}$ **then**
12 $\quad\quad \underline{Collected \leftarrow ValidNonZero}$
13 $\quad$ **else**
14 $\quad\quad \underline{Collected \leftarrow Valid}$
15 $\quad$ **end**
16 $\quad$ Sort T in decreasing order of $sat(uncov(t))$
17 $\quad interrupted \leftarrow FALSE$
18 $\quad$ **for** $t \in T$ **do**
19 $\quad\quad$ **if** $(Collected \wedge uncov(t)) \neq FALSE$ **then**
20 $\quad\quad\quad Collected \leftarrow Collected \wedge uncov(t)$
21 $\quad\quad$ **end**
22 $\quad\quad$ **if** $numNodes(Collected) > sizeThrsld$ **then**
23 $\quad\quad\quad interrupted \leftarrow TRUE$
24 $\quad\quad\quad$ Break
25 $\quad\quad$ **end**
26 $\quad$ **end**
27 $\quad$ **if** *interrupted* **then**
28 $\quad\quad$ **for** $i = 1$ *to* n **do**
29 $\quad\quad\quad candt_i \leftarrow$`randSat`$(Collected, \underline{curWgts})$
30 $\quad\quad\quad newCov_i \leftarrow$`newlyCovered`$(candidate_i, T)$
31 $\quad\quad$ **end**
32 $\quad\quad chosen \leftarrow \{candt_i | \forall j. newCov_i \geq newCcov_j\}$
33 $\quad$ **else**
34 $\quad\quad chosen \leftarrow$`randSat`$(Collected, \underline{curWgts})$
35 $\quad$ **end**
36 $\quad$`appendResult`$(chosen)$
37 $\quad \underline{\texttt{updateWeights}(curWgts)}$
38 $\quad$ **for** $t \in T$ **do**
39 $\quad\quad uncov(t) \leftarrow uncov(t) \wedge \neg chosen$
40 $\quad\quad$ **if** $uncov(t) = FALSE$ **then**
41 $\quad\quad\quad T \leftarrow T \setminus t$
42 $\quad\quad$ **end**
43 $\quad$ **end**
44 **end**

Algorithm 2: BDD-Based CTD with Support for Weights

conjunction of the valid tests and a BDD representing only the non-zero weighted values (Lines 4 – 10 and 12). This is performed only if there exists some test that satisfies this non-zero weights constraint and covers some new assignment to at least one tuple. If no such test exists, then the algorithm proceeds as before, with all valid tests (Line 14).

5. EXPERIMENTAL RESULTS

5.1 Test Space Construction

Representing a test space using BDDs is clearly much more efficient than explicit representation. Table 1 considers several test space instances, of different sizes and different complexity of restrictions, and compares the time required for constructing the BDD representation for them (i.e., building the *Valid* and *Invalid* BDDs) to the time needed for constructing an explicit representation. Whenever the explicit representation required more that 10 minutes or more than 128 MB of memory, a N/A value is given in the table. The table also details for each test space the total number of value combinations in it, and the percentage of tests that are valid.

It is clear from the table that only small test spaces, up to about two million combinations, can be represented explicitly, while the BDD representation easily scales up even to very large test spaces with very complex restrictions.

5.2 Projection

An important feature of the FoCuS tool is that of projection – the user chooses a subset of the parameters, and the Cartesian product of these parameters is displayed, along with the validity of each entry. Even though this Cartesian product may be relatively small, in order to compute the validity of each entry (valid, invalid or partially valid), the entire Cartesian product of all parameters should be considered. Therefore, as in the test space representation above, explicit representation of the test space fails to compute the projection of large test spaces, even to a small set of parameters.

Using BDDs, however, this operation is very efficient. In all test space instances given in Table 1, projecting them on one, three and five parameters is computed in less than 100 milliseconds.

5.3 Combinatorial Test Design

We compare the results of the BDD-based combinatorial test design algorithm to those of other known algorithms on two sets of test spaces. The first is a set of twenty real-life test space instances generated by or for our customers. The test spaces range in the number of parameters, number of restrictions, and the percentage of tests that are valid out of the entire set of possible tests. The details of the test spaces are given in Table 2, which lists for each test space the number of parameters in it, the total number of value combinations in it, the percentage of tests that are valid, and the number of restrictions. Note that in general, FoCuS allows restrictions to be given as any Boolean expression in Java syntax. We translate each restriction into DNF form, and the number of restrictions given in the table is the total number of conjunctive clauses in the restrictions. As mentioned in Section 3.1, the test spaces represent testing problems from different domains, such as telecommunication, healthcare, storage and banking, and for testing different aspects

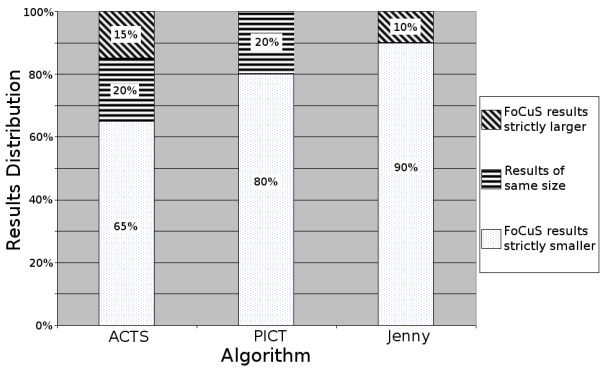

Figure 4: Summary of CTD results in FoCuS compared to other algorithms, on real-life test spaces

of the system, such as data manipulation and integrity, protocol validation and user interface testing. The test space definitions for these problems, as well as the solutions found by FoCuS, can be downloaded from [8].

The size of the solution found by FoCuS for pairwise coverage in each test space is compared to that of three other algorithms[2] - PICT [9], ACTS [25] and Jenny [18]. Table 2 lists the size of the solutions found by each algorithm. Figure 4 summarizes these numbers by showing the percentage of computations for which FoCuS found strictly smaller, equal or strictly larger results compared to those by each of the other tools. FoCuS generated strictly smaller results than Jenny for 90% of the test spaces, strictly smaller than PICT for 80% of them, and strictly smaller than ACTS for 65%[3].

The results obtained by FoCuS are also compared in Table 3 to those of other known algorithms for pairwise coverage of some standard benchmarking test spaces. Note that these instances do not contain restrictions at all, thus are much less representative of real-life test spaces. On these test spaces, FoCuS generates results comparable to those of other tools.

5.4 Weights

Finally, we explore the degree to which the results generated by FoCuS are compliant with weight requirements. In this experiment, each of the 26 test spaces presented in Tables 2 and 3 was given to the tool five times. In each run, a single parameter was chosen randomly, and for each of its values, weights were chosen randomly, between 1 and 10. The distribution of these values in the result of the algorithm was compared to the required distribution by computing the KL-divergence [20] of the result with respect to the requirement. KL-divergence between distributions is commonly used as a measure of similarity between distributions. The closer the distributions, the smaller the value is.

Figure 5 plots the cumulative histogram of the KL values in all the runs (total of $26 \times 5 = 130$ runs). The cumulative histogram is plotted for FoCuS and PICT (to our knowledge, the only tool other than FoCuS that supports weights). The cumulative histograms are also compared to

[2]To our knowledge, these are the only tools that are both freely available and support restrictions.

[3]For five test spaces, ACTS failed to find a solution within twelve hours.

Test Space Name	Total Number of Combinations	Valid Tests Percentage	Test Space Construction Time (Milliseconds)	
			BDDs	Explicit
Concurrency	32	25%	140	0
Banking1	324	65.43%	187	0
CommProtocol	7,168	5.08%	250	391
Healthcare2	93,312	37.50%	234	1187
3^{13}	1,594,323	100%	16	29110
NetworkMgmt	2,200,000	51.48%	141	N/A
Healthcare4	1.87×10^{16}	20.16%	156	N/A

Table 1: Time for constructing a test space

Test Space Name	Num. Params	Number of Value Combinations	Percentage of Valid Tests	Num. Restrictions	ACTS [25]	PICT [9]	Jenny [18]	FoCuS
Concurrency	5	32	25%	7	5	6	5	6
Storage1	4	120	20.83%	95	36	18	19	17
Banking1	5	324	65.43%	112	N/A	18	16	14
Storage2	5	486	100%	0	19	19	20	18
CommProtocol	11	7,168	5.08%	128	N/A	19	19	16
SystemMgmt	10	12,960	7.5%	17	19	19	19	16
Healthcare1	10	17,280	25%	21	30	31	32	30
Telecom	10	46,080	39.84%	21	31	32	32	30
Banking2	15	65,536	62.5%	3	10	14	14	13
Healthcare2	12	93,312	37.5%	25	18	18	20	18
NetworkMgmt	9	2,200,000	51.48%	20	132	129	126	115
Storage3	15	9,216,000	2.88%	48	52	54	55	52
ProcessorComm1	15	2.39×10^7	33.51%	13	32	30	28	29
Services	13	1.04×10^8	1.22%	388	N/A	111	117	102
Insurance	14	2.60×10^9	100%	0	527	527	529	527
Storage4	20	2.29×10^{11}	63.08%	24	134	130	138	130
Healthcare3	29	1.47×10^{12}	1.45%	31	38	40	42	35
ProcessorComm2	25	6.97×10^{12}	0.006%	125	32	36	36	33
Storage5	23	7.48×10^{13}	1.2%	151	N/A	235	237	226
Healthcare4	35	1.87×10^{16}	20.16%	22	N/A	53	58	47

Table 2: Comparing CTD results with known CTD algorithms on real-life test spaces

Test Space	AETG [10]	IPO [28]	TConfig [30]	CTS [17]	Jenny [18]	DDA [7]	AllPairs [1]	PICT [9]	ACTS [25]	FoCuS
3^4	9	9	9	9	11	?	9	9	12	10
3^{13}	15	17	15	15	18	18	17	18	21	20
$4^{15}3^{17}2^{29}$	41	34	40	39	38	35	34	37	33	37
$4^{1}3^{39}2^{35}$	28	26	30	29	28	27	26	27	28	30
2^{100}	10	15	14	10	16	15	14	15	16	15
10^{20}	180	212	231	210	193	201	197	210	215	259

Table 3: Comparing CTD results with known CTD algorithms on standard test spaces

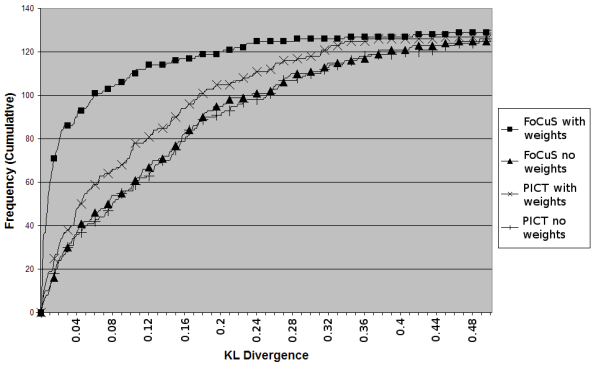

Figure 5: Cumulative histogram of KL divergence in result vs requirement for FoCuS and PICT

those achieved by the tools without specifying the weight requirement. Clearly, FoCuS returned results that are distributed much closer to the requested distribution. For example, on 101 of the runs of FoCuS the KL divergence was smaller or equal to 0.06, while PICT gave only 59 results with such divergence.

A possible explanation for these results is that in each iteration, our algorithm chooses a test from a relatively heterogeneous set of tests, in which many values are unfixed. This leaves many degrees of freedom for the selection of a test that maintains the required distribution.

We note that in FoCuS, introducing weight requirements caused the results to be larger by six percent on average, while no such behavior was observed in PICT.

6. CONCLUSIONS AND FUTURE WORK

Combinatorial test design is an effective test planning technique that reveals faulty feature interaction in a given system. In this paper we presented a Cartesian-product based methodology for constructing a test space as an input for a CTD algorithm, along with a BDD-based tool support for it. We also introduced a BDD-based algorithm for solving the CTD problem itself, by efficient subset selection from the entire set of valid tests, and showed how it can be extended to effectively support also value distribution requirements, specified by assigning weights to values. Finally, we explored some experimental results, based on complex real-life test design tasks of our customers, taken from a diverse range of domains and representing different aspects of systems. These results show how the BDD implementation dramatically increases the size of the test spaces that may be supported for the Cartesian-product based methodology, and also superior results of our CTD algorithm over other known algorithms on most real-life problems.

We would like to extend the Cartesian-product based methodology and technology for test space construction in order to further assist the user in constructing and reviewing the test space. One possible direction is to automatically map between illegal combinations, or partially legal ones, and the restrictions that exclude them. This can be a highly useful debugging aid for the tester, that will help in reviewing the test space, understanding it, and correcting restrictions in order to make certain combinations valid again.

We would also like to extend our methodology and technology to support incremental modifications of the system under test, or of the test design requirements. Such an extension would include assistance in identifying and applying required changes to the test space, as well as support in the update of the test plan. In addition, we would like to support more complex structures in the test space definition, on top of the existing representation as parameters, values and restrictions.

A well known problem with BDDs is their sensitivity to the order of variables. The representation of a given Boolean function with BDDs may be very efficient using one order of the variables, and very inefficient using another order. Therefore, correctly choosing the order of variables, though computationally hard [2], is a crucial point in BDD implementations. Exploring different heuristics for variable ordering in the BDD-based representations and algorithms presented in this paper is left as future work.

Finally, in this work, we explore how BDDs can be used in order to effectively support weight requirements in CTD problems. In the future, we plan to study how other advanced features of CTD may benefit from our BDD implementation of CTD.

7. REFERENCES

[1] Allpairs website. http://www.mcdowella.demon.co.uk/allPairs.html.

[2] B. Bollig and I. Wegener. Improving the Variable Ordering of OBDDs is NP-Complete. *IEEE Transactions on Computers*, 45:993–1002, Sept. 1996.

[3] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[4] A. Calvagna and A. Gargantini. A Logic-Based Approach to Combinatorial Testing with Constraints. In *Proc. 2nd International Conference on Tests and Proofs (TAP'08)*, pages 66–83. Springer-Verlag, 2008.

[5] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction Testing of Highly-Configurable Systems in the Presence of Constraints. In *Proc. 2007 International Symposium on Software Testing and Analysis (ISSTA'07)*, pages 129–139. ACM, 2007.

[6] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn. Constructing Test Suites for Interaction Testing. In *Proc. 25th International Conference on Software Engineering (ICSE'03)*, pages 38–48. IEEE Computer Society, 2003.

[7] C. J. Colbourn, M. B. Cohen, and R. Turban. A Deterministic Density Algorithm for Pairwise Interaction Coverage. In *IASTED Conference on Software Engineering*, pages 345–352. IASTED/ACTA Press, 2004.

[8] CTD Benchmark Files. http://researcher.watson.ibm.com/researcher/files/il-ITAIS/ctdBenchmarks.zip.

[9] J. Czerwonka. Pairwise Testing in Real World. In *Proc. 24th Pacific Northwest Software Quality Conference (PNSQC'06)*, pages 419–430, 2006.

[10] M. L. Fredman D. M. Cohen, S. R. Dalal and G. C. Patton. The AETG System: An Approach to Testing Based on Combinatorial Design. *IEEE Transactions on Software Engineering*, 23:437–444, 1997.

[11] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton,

C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-Based Testing in Practice. In *Proc. 21st International Conference on Software Engineering (ICSE'99)*, pages 285–294, 1999.

[12] O. Grumberg E. M. Clarke and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[13] T. Erlz. *SOA Principles of Service Design*. Prentice Hall PTR, 2007.

[14] FoCuS website. http://researcher.watson.ibm.com/-researcher/view_project.php?id=1871.

[15] M. Grindal, J. Offutt, and S. F. Andler. Combination Testing Strategies: A Survey. *Software Testing, Verification, and Reliability*, 15:167–199, 2005.

[16] A. Hartman. Software and Hardware Testing using Combinatorial Covering Suites. In *Graph Theory, Combinatorics and Algorithms: Interdisciplinary Applications*, pages 266–327. Springer-Verlag, 2005.

[17] A. Hartman and L. Raskin. Problems and Algorithms for Covering Arrays. *Discrete Mathematics*, 284(1-3):149–156, 2004.

[18] Jenny website. http://burtleburtle.net/bob/math/jenny.html.

[19] D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software Fault Interactions and Implications for Software Testing. *IEEE Transactions on Software Engineering*, 30:418–421, 2004.

[20] S. Kullback and R. A. Leibler. On Information and Sufficiency. *Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[21] S. Malik, A.R. Wang, R.K. Brayton, and A. Sangiovanni-Vincentelli. Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment. In *IEEE International Conference on Computer-Aided Design (ICCAD'88)*, pages 6–9. IEEE, 1988.

[22] S. Minato. Graph-Based Representations of Discrete Functions. *Representation of Discrete Functions*, pages 1–27, 1996.

[23] K. J. Nurmela. Upper Bounds for Covering Arrays by Tabu Search. *Discrete Applied Mathematics*, 138:143–152, 2004.

[24] Pairwise testing website. http://www.pairwise.org.

[25] Y. Lei R. Kuhn and R. Kacker. Practical Combinatorial Testing: Beyond Pairwise. *IT Professional*, 10:19–23, 2008.

[26] Spec explorer. http://msdn.microsoft.com/en-us/-library/ee620448.aspx.

[27] R. Brayton T. Kam, T. Villa and A. Sangiovanni-Vincentelli. Multi-Valued Decision Diagrams: Theory and Applications. *Multiple-Valued Logic*, 4(1-2):9–62, 1998.

[28] K.C. Tai and Y. Lie. A Test Generation Strategy for Pairwise Testing. *IEEE Transactions on Software Engineering*, 28:109–111, 2002.

[29] A. W. Williams and R. L Probert. A Measure for Component Interaction Test Coverage. In *Proc. ACSI/IEEE International Conference on Computer Systems and Applications (AICCSA'01)*, pages 304–311, 2001.

[30] Alan W. Williams. Determination of Test Configurations for Pair-Wise Interaction Coverage. In *Proc. IFIP TC6/WG6.1 13th International Conference on Testing Communicating Systems: Tools and Techniques (TestCom'00)*, pages 59–74. Kluwer, B.V., 2000.