Simplified Modeling of Combinatorial Test Spaces

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

Aviad Zlotnick
IBM, Haifa Research Lab
Haifa University Campus
Haifa, 31905, Israel
aviad@il.ibm.com

***Abstract*—Combinatorial test design (CTD) is an effective test planning technique that reveals faults that result from feature interactions in a system. The test space is manually 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.**

In many real-life testing problems, the relationships between the different test parameters are complex. Thus, precisely capturing them by restrictions in the CTD model might be a very challenging and time consuming task. From our experience, this is one of the main obstacles in applying CTD to a wide range of testing problems. In this paper, we introduce two new constructs to the CTD model, counters and value properties, that considerably reduce the complexity of the modeling task, allowing one to easily model testing problems that were practically impossible to model before. We demonstrate the impact of these constructs on two real-life case studies.

I. Introduction

As software systems become increasingly complex, verifying their correctness is more challenging. The introduction of service-oriented architectures (SOA) [7] 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 interactions) of every t parameters, where t is a user input. In general, one can require different levels of interaction for different subsets of parameters. 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 [14], [6]. Other experimental work has shown that typically 100% of bugs can be revealed by covering the interaction of between 4 and 6 parameters [11].

We applied CTD to different domains such as hardware interoperability testing, and testing of software in the healthcare, financial, and other domains, and found that the effort and expertise that is needed to define correctly the set of restrictions that captures the valid test space is one of the main deployment obstacles. The relationships between the different elements of the test space can be highly complex, resulting in a labor-intensive process of identifying and specifying the restrictions that capture them. Existing work on restrictions in combinatorial models concentrates on the ability to state complex restrictions, preferably as propositional formulas or predicates [5], [6], [3]. A survey of restriction handling techniques in existing tools is given in [4], recognizing the importance of the ability to support full restrictions without remodeling of the parameters or explicit enumeration of all invalid tests. Restrictions can also be captured using classification tree methods [8]. Tree-based descriptions for CTD models are useful for capturing dependencies in hierarchical test spaces, but the mere usage of trees does not address more complex relationships between the different elements of the test space.

In this paper, we suggest a different approach for handling complex restrictions in combinatorial models. Specifically, we introduce two new constructs to the CTD model. We demonstrate on real-life examples how these constructs can significantly reduce the number and the complexity of the restrictions that need to be stated. The first construct we introduce is counter parameters. A counter is a special type of parameter. It counts in each test the number of appearances of specific values for specific parameters, as defined by the user. The second construct is value properties. A property is associated with a parameter, and for each value of the parameter, a value is assigned to the property. The property values represent additional aspects that are not expressed in the original parameter values.

For each of the two constructs, we present a real-life case study demonstrating the significant reduction that is achieved in the complexity of the combinatorial model. In addition, we demonstrate how these constructs can reduce not only the number and complexity of the restrictions, but also that of the CTD coverage requirements. Finally, we show how these constructs can be efficiently implemented using Binary Decision Diagrams (BDDs) [2], which our CTD tool uses as its internal data structure [13]. These constructs can probably be efficiently supported by other approaches as well, such as SMT-based approaches [3].

Special constructs for modeling complex situations in combinatorial models, beyond the basic parameters, values and restrictions, have been introduced before. [12] describes the notions of compound values, auxiliary aggregates, and group fields. A compound is a set of values for parameters that the user defines when the interaction coverage is not required within the values in the compound, but is needed with other parameters. An auxiliary aggregate captures parameters and values that are common between different test spaces, and is reused among them instead of redefining the common elements for each test space. Field groups are used in order to support optional groups of values. If any one field from the group is present, then all fields from the group must be present; however, the entire group is optional. In [5], hierarchies of parameters may be defined. Once defined, parameters at lower levels are t-wise combined first. The result of lower level combinations are then used for creating combinations with parameters at higher levels of the hierarchy. In [10], the notion of sub-attributes is used in order to describe compound parameters, that is, parameters that are associated with sub-parameters, and alternatives are discussed regarding how to treat them at test case generation phase.

Most of these constructs are used in order to simplify the interaction coverage requirements in the CTD algorithm. Auxiliary aggregates and field groups are also used in order to reduce the number of restrictions. The latter construct can also reduce the complexity of the restrictions. The constructs that we introduce in this paper address other modeling challenges that still require simplification.

In [1], a test specification language called TSL is introduced, and a concept somewhat similar to properties is presented (called categories). The context in [1] is automatic generation of executable test scripts, rather than CTD. The tester identifies the significant categories for each parameter or environment, and splits each category into a set of mutually exclusive choices, from which the value of the parameter or environment condition is assigned. In the context of automatic test generation, the categories for parameters are usually parameters of an actual function or command, and the categories for environments are usually features of the test environment. The categories are used to determine the result of the test, but one cannot express arbitrary restrictions between different categories, as opposed to the properties proposed in this work. Another difference is that as opposed to TSL categories that represent the concrete value of the parameter, the properties we propose for CTD have a separate value from the parameter they are assigned to, representing the different abstract aspect that is captured by the property than is captured by the parameter itself.

In the following we present in Section II background on CTD and BDDs. Sections III and IV introduce counters and value properties, respectively, including a case study that is modeled with and without the introduced construct, and an efficient implementation for the construct. Finally, Section V draws our conclusions and future research directions.

II. Background

A. Combinatorial Test Design (CTD)

Combinatorial Test Design (CTD) is a well-known, powerful technique for functional test planning, in which all value combinations of size t of the parameters that describe a system are tested.

The technique calls for identifying a set of parameters, along with corresponding values. Typically, restrictions on combinations of values that are not to appear in a test are also defined, in some formal language. For example, in IBM's CTD tool developed in our group [9], restrictions can be any Boolean expression in Java. We refer to these parameters, values and restrictions as an *input model*, or a *model*. A test plan is a set of tests, where a test is represented by a tuple that assigns a value to each parameter.

Given a model, and an integer t, CTD generates a test plan in which for every t parameters, every valid combination of values for them appears in at least one test. A combination of values is *valid* if it is not excluded by any restriction. We refer to the above integer t as the *level of interaction*. In general, one can require different levels of interaction for different subsets of parameters. We generally refer to these requirements as *interaction coverage requirements*, or simply *coverage requirements*. A common special case is one in which $t = 2$. This case is usually referred to as *pairwise testing*.

B. Binary Decision Diagrams (BDDs)

Our tool uses Binary Decision Diagrams (BDDs) [2] as its internal data structure, as detailed in [13]. For the purpose of this paper, it suffices to consider Binary Decision Diagrams as an efficient data structure for representing sets of Boolean vectors (in our case, sets of tests in a given combinatorial model), and for performing operations on them. In this setting, a set of tests is represented by a single BDD. Non-Boolean parameters are supported by applying a standard reduction from multi-valued variables to sets of Boolean ones.

Given an input model, the tool constructs the BDD that represents all valid tests by parsing the restrictions and applying appropriate set operations. This set is then used as the basis for the CTD computation, as well as for computing other analyses of the model.

In general, the tool uses BDDs in which the variables represent the parameters in the model, and constructs the set of valid tests as follows. Each restriction, $Rest_i$, is parsed and a BDD $Allowed_i$ is constructed such that it captures the set of tests allowed by this restriction. The set of all valid tests is then simply the intersection of all $Allowed_i$ sets. More details may be found in [13].

III. Counters

A. Example

Consider testing an upgrade process for a virtualized server. The upgrade process traverses the host and all the virtual machines in parallel. For each machine it traverses its various components, upgrading those that are out of date. The host components for which upgrade is performed can include for example the host operating system, virtualizer, network card firmware, disk firmware, etc. Each virtual machine has one of several operating system types, and each type can have multiple versions. One of the versions is marked as the preferred version, to which the operating system is upgraded. The upgrade process is triggered when the installed operating system is not at the preferred version at least on half of the virtual machines.

To test the upgrade process, we would like to make sure it operates correctly with different combinations of component types on the different virtual machines and on the host. This is a classical application of CTD. Parameters can be defined for the possible operating systems on the virtual machines, as well as for the possible components of the host. Assume that we have 10 virtual machines and a single host. The model of the test space will contain 10 matching parameters, each specifying the types of operating systems that the corresponding virtual machine can have. For simplicity, we use the same parameter to also capture the possible versions of the operating system, and assume that the sets of possible operating systems are identical for all virtual machines. For example, the model can contain 10 parameters, named `VM1` to `VM10`, each with the values `RedHat5.7`, `RedHat6.1`, `AIX6.1.3`, and `AIX6.1.6`, where `RedHat6.1` and `AIX6.1.6` are marked as the preferred versions. Additional parameters will capture the upgraded host components.

However, it is very difficult to capture the requirement that an out of date version of the operating system is installed on at least half of the virtual machines. This requires explicitly excluding all value combinations where less than half of the operating systems are assigned with a non-preferred value. In our example, containing 10 machines, each with 2 preferred values and 2 non-preferred values, there are 653,312 such combinations. A much more efficient strategy than explicitly excluding all these combinations, would be to choose 5 parameters out of the 10, and force each of them to a non-preferred value. An example of such a restriction is as follows:

`VM1` *neq* `''AIX6.1.6'' && VM1` $\neq$ `''RedHat6.1'' && VM2` $\neq$ `''AIX6.1.6'' && VM2` $\neq$ `''RedHat6.1'' && ...&& VM5` $\neq$ `''AIX6.1.6'' && VM5` $\neq$ `''RedHat6.1''.`

This approach would still require 252 such restrictions, and in general $\binom{n}{k}$ restrictions, where n is the number of virtual machines, and k is the threshold number for upgrade. Clearly, it is practically impossible to model this example, that is based on a real-life case, using only parameters, values and restrictions. So would be the general case in which restrictions refer to the number of occurrences of specific values or groups of values in a test, as will be further explained in the following.

B. Introducing Counters

To enable modeling cases as described above, we introduce a new type of parameter, named *counter*. A counter is defined by the values of other parameters that it counts, and is automatically evaluated in each test to the number of occurrences of these values. It can then be referred to in the restrictions and in the coverage requirements. In the example above, one could define a counter for the preferred values in any of the `VM` parameters, and then replace the 252 restrictions with a single, simple restriction, excluding the case where the value of the counter is greater than 4, i.e., exclude `counter > 4`.

Thus, by adding one counter parameter to the model, the number of restrictions that need to be manually entered is reduced from 252 to 1, and in general from $\binom{n}{k}$ to 1. This turns the modeling effort of this type of models (where counting is required) from practically infeasible to relatively simple.

A further simplification (though not as dramatic as the one described above) is achieved by allowing the user to define the counter on “any parameter”, rather than separately specifying all the parameters for which values are counted. In this case, the union of values of all parameters is displayed, and the user selects from it the values to be counted. Whenever a selected value appears in a test, regardless of the associated parameter, it will be counted by the counter. In the example above, the user can define a counter on “any parameter” with values `RedHat6.1` and `AIX6.1.6`.

Counters are useful for any case in which there is importance to the number of occurrences of specific values or groups of values in a test. For example, any case in which parameters are divided to groups, and we want to compare the number of specific values from each group, or refer to a threshold on the number of specific values in a specific group. From our experience in real-life deployment of CTD, this is quite a common scenario.

The notion of counters can be extended to the general case of *auxiliary parameters*. These are parameters that do not define the test, but are rather a function of the defining parameters, and are used in order to more easily capture different characteristics of the test. For example, one could define an auxiliary parameter to be the sum of the values of other numeric parameters. Another example is an auxiliary parameter that counts in each test the number of duplicate values among a given set of parameters.

A naïve implementation of counters would be to treat them as regular parameters, and translate the function that defines them into additional restrictions on the valid tests. In Section III-D, we propose a much more efficient implementation for counters, that enables using them in real-life cases without exhausting the computation resources.

C. Counters and Coverage Requirements

As mentioned above, the counter can be referred to not only in the restrictions but also in the coverage requirements. For example, one can request to cover the case where no virtual machine is installed with a preferred operating system version, only one is installed, and so on up to the threshold of four virtual machines. Similarly, one can request to cover combinations of the counter values with the values of other parameters of the test space.

D. Implementing Counters

A naïve implementation of counters would be to treat them as regular parameters, and introduce restrictions that ensure that the correct value is given to each counter, as a function of the other parameters. This approach is inefficient – it introduces redundant parameters that enlarge the cross product to be handled, as well as numerous restrictions in order to make sure that the counters capture the intended function correctly.

Instead, we use our tool's symbolic BDD representation in order to support counters much more efficiently. The BDD representation requires two things: 1) to define the variables that participate in the BDD, and 2) to compute the BDD that captures the set of valid tests. This approach is extended to support counters as follows.

1) BDD variables: The variables of the BDDs are still the regular parameters (we hereby refer to parameters which are not counters as *regular parameters*). Counter parameters need not be represented directly as variables in the BDDs, since their values are determined by those of the regular ones.

2) The set of valid tests: As explained in Section II-B, given a model, the BDD approach requires constructing a single BDD that captures the set of valid tests. This is achieved by constructing for each restriction, $Rest_i$, the BDD $Allowed_i$ that captures the set of tests allowed by this restriction, and intersecting these sets. Clearly, for restrictions that do not refer to counter parameters, the computation remains exactly as before – the restriction is parsed, and the $Allowed_i$ BDD is constructed directly from it. For restrictions that do refer to counters, however, the tool has to construct a BDD on the regular parameters that correctly captures the restriction on the counters.

A possible approach to this would be to first translate the restriction into one or more restrictions on the regular parameters. A better solution, however, is to utilize BDD operations in order to construct the $Allowed$ set directly from the restriction. First, for each relevant value of the counter, a BDD is constructed that captures all tests for which the counter receives this value (for example, for the value 4 of the preferred versions counter, this BDD would capture all tests in which exactly four operating systems are in a preferred version). This is performed by recursively iterating over the values of the relevant regular parameters, and gradually constructing the BDDs for the counter values. These BDDs can then be used in order to construct the restriction's BDD. For example, the restriction according to which at most four operating systems are in preferred versions is captured by the union of the BDDs corresponding to values four and below.

Note that this implementation is superior to the naïve one both in memory consumption (BDDs are used all along the computations, rather than explicit listing of all relevant combinations) and in time (standard techniques such as caching of intermediate results may be employed).

IV. Properties

A. Example

Consider testing a Storage Area Network (SAN) system, consisting of a host, an operating system and a host bus adapter (HBA), where each component can be one of several types. In order to test such a configuration, one has to ensure that components of different types work together correctly. CTD is a natural fit to constructing a test plan in such a setting – one would define parameters capturing the hardware architecture on which the server is running (the `Server` parameter), the operating system it is using (the `OS` parameter), and the HBA type (the `HBA` parameter), and require, for example, coverage of pairwise interactions.

However, a serious obstacle to correctly applying CTD to the above example arises when one tries to capture the constraints between the parameters. Some operating systems cannot run on certain architectures, e.g., Windows-based operating systems cannot run on PowerPC platforms, while AIX ones cannot run on x86 architectures. Similarly, some

HBAs can be used on one platform and not on another, and some may have drivers for one OS family and not for another. Note that there may be several different Windows-based operating systems, several different x86-based platforms, etc., and the constraints should be correctly captured for all of them.

The example above is based on a real-life user scenario, in which there were 4 different server platforms (two PowerPC-based, and two x86-based), 17 operating system versions (two Windows-based, three AIX-based, one virtualized OS, and the rest Linux-based), and 11 different HBAs, from three different manufacturers. Explicitly excluding all invalid combinations would require reviewing 299 pairs (68 of them matching operating systems and architecture, 187 matching HBAs and operating systems, and 44 matching HBAs and architecture), and carefully identifying the 34 that should be excluded. This is clearly a tedious and highly error-prone process.

B. Introducing Properties

In order to overcome the limitation demonstrated by the above example, we introduce properties to CTD input models. The main idea in the properties features is that one may define a property for a parameter, and then assign a value to the property for each value of the parameter. The properties can then be used for defining restrictions and coverage requirements. In our example, one could define an `Architecture` property for the `Server` parameter. The two PowerPC servers would have a `PowerPC` value for this property, while the two x86-based ones would have `x86`. The `HBA` parameter would have four Boolean properties – `XCompatible`, `PCompatible`, `AIXCompatible`, and `WinCompatible`, representing for each HBA, whether it is compatible with x86 architecture, PowerPC architecture, AIX OS and Windows OS, respectively. Finally, the `OS` parameter would have an `OSFamily` parameter with possible values `Windows`, `AIX` and `Linux`.

Once properties are defined, one can refer to them in the restrictions. For example, the expression `Server.Architecture` refers to the architecture of the specific server. Consider the restriction that Windows OS cannot run on PowerPC platforms. Previously, we were compelled to explicitly list all combinations of a Windows-based operating system and a PowerPC-based server, and exclude them. This is now replaced by a single, much more general, restriction that excludes: `Server.Architecture == ``PowerPC'' && OS.OSFamily == ``Windows''`. This restriction instructs the modeling tool to exclude any combination of `Server` and `OS`, in which the value of the `Architecture` property of the server is equal to `PowerPC`, and the value of `OSFamily` is `Windows`. Clearly, the new form of the restriction is significantly more readable (thus easier to review), more general (thus preserves time and reduces the risk of introducing errors), and more maintainable (e.g., if a new Windows-based OS is introduced, this OS is immediately covered by the restriction).

The entire set of 34 restrictions can be replaced by the following six restrictions:

- `Server.Architecture == ``PowerPC'' && OS.OSFamily == ``Windows''`
- `Server.Architecture == ``x86'' && OS.OSFamily == ``AIX''`
- `Server.Architecture == "PowerPC" && ! HBA.PCompatible`
- `Server.Architecture == "x86" && ! HBA.XCompatible`
- `OS.OSFamily == "AIX" && ! HBA.AIXCompatible`
- `OS.OSFamily == "Windows" && ! HBA.WinCompatible`

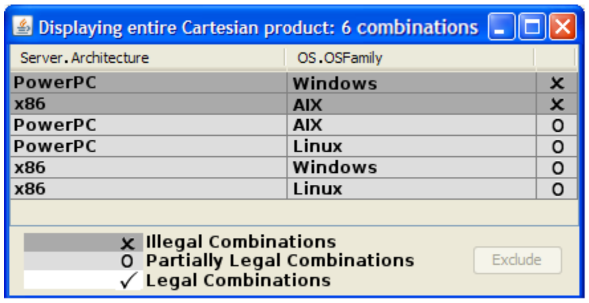
C. Properties and Projections

Our tool supplies a convenient method of defining restrictions by using *projections*. A projection is a Cartesian product of a given set of parameters (typically a subset of all parameters), in which each combination is marked as one of the following: a combination is *valid* if any extension thereof to a complete test (i.e., any possible assignment to those parameters not participating in the projection) is valid. A combination is *invalid* if any extension thereof is invalid. Finally, a combination is *partially valid* if it is neither valid nor invalid (i.e., there exists an extension that is valid, and one that is invalid).

From our experience, projections are very effective both for defining the model, and for reviewing it. For reviewing, one considers projections of different subsets of parameters, and reviews the possible combinations in each. By considering each combination individually, along with its validity, one can easily verify whether the correct restrictions have been defined for this subset of parameters. This review methodology is effective both for identifying omissions (i.e., combinations that should be valid, but are not), as well as verifying correctness (i.e., that the combinations marked valid are indeed the correct ones).

Projections are also a convenient user interface for defining restrictions – instead of writing explicit expressions that exclude certain combinations, one may simply mark them in the appropriate projection, and click an “Exclude” button. The tool then automatically adds restrictions that exclude the selected combinations.

Combining the properties feature with that of projections is an important step for the usability of properties. Much like “standard” projections, the user may, in addition to selecting parameters to project the model on, also select parameter properties for the projection. For example, Figure 1 shows


Displaying entire Cartesian product: 6 combinations

Server.Architecture	OS.OSFamily	
PowerPC	Windows	✗
x86	AIX	✗
PowerPC	AIX	O
PowerPC	Linux	O
x86	Windows	O
x86	Linux	O

✗ Illegal Combinations
O Partially Legal Combinations
✓ Legal Combinations

Figure 1. A projection of the SAN system example on two properties

the projection on properties `Server.Architecture` and `OS.OSFamily`. Note that this projection is an abstraction of the projection on the corresponding parameters, i.e., each row in this Cartesian product represents multiple rows in the Cartesian product of the corresponding parameters. For example, the row in which the architecture is `x86` and OS family is `Windows` represents all combinations of an x86 architecture and a Windows OS – a total of four concrete combinations.

The validity of rows is similar to that in projections on regular parameters. In our example, the first two restrictions above make two rows in the projection invalid. The other rows are partially valid, representing the fact that some extensions thereof to the entire set of parameters are valid, while others are invalid. Note that in this example, the projection is presented after the restrictions were defined (therefore, the relevant rows are already marked invalid), but it could have also been used for defining these restrictions, by selecting the relevant rows and excluding them from the model.

D. *Properties and Coverage Requirements*

Thus far, we have discussed the advantage of using properties for correctly and easily capturing the input model for the CTD algorithm. Another advantage in introducing properties into the modeling of combinatorial test spaces is when defining the interaction coverage requirements. In our example, one can now refine the interaction coverage requirements between the operating system and server, and easily require that each operating system is tested on each server architecture (i.e., at least once on an x86-based machine, and at least once on a PowerPC one), rather than on each concrete server. Enabling such a refinement of the coverage requirements, by referring to a grouping of values rather than to all values individually, can lead to a significant reduction in the size of the test plan.

E. *Implementing Properties*

The same concepts applied for supporting counters, discussed in Section III-D, can be applied for supporting properties. A naïve implementation is to treat them as regular parameters, and add restrictions accordingly. In our BDD-based optimized implementation, the variables of the BDDs remain the parameters in the model, regardless of properties. For each value of a property, a BDD is constructed that captures all tests in which the corresponding parameter gets a corresponding value. For example, a BDD is constructed that represents all tests in which the operating system is Windows-based. Restrictions can now be parsed as usual, and whenever they refer to properties, the appropriate BDD is used.

V. Future Work

In this paper, we introduced two new constructs to the modeling of combinatorial test spaces. Using real-life case studies, we showed how these constructs dramatically reduce the manual effort of defining the test space, turning it from practically impossible to relatively simple. We plan to conduct an empirical study to systematically analyze the usage of these constructs and identify points for improvement. In addition, we plan to investigate additional currently unanswered modeling challenges that we encountered, and suggest suitable solutions and optimized implementations. Finally, we plan to investigate combining different constructs to achieve a more powerful specification mechanism for combinatorial test spaces.

References

[1] M. Balcer, W. Hasling, and T. Ostrand. Automatic generation of test scripts from formal test specifications. *SIGSOFT Softw. Eng. Notes*, 14:210–218, 1989.

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

[3] 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.

[4] 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.

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

[6] 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. ACM, 1999.

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

[8] M. Grochtmann and K. Grimm. Classification trees for partition testing. *SIGSOFT Softw. Eng. Notes*, 3:6382, 1993.

[9] IBM's CTD tool website. http://researcher.watson.ibm.com/-researcher/view_project.php?id=1871.

[10] R. Krishnan, S. Murali Krishna, and P. Siva Nandhan. Combinatorial testing: learnings from our experience. *SIGSOFT Softw. Eng. Notes*, 32:1–8, 2007.

[11] 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.

[12] C. Lott, A. Jain, and S. Dalal. Modeling requirements for combinatorial software testing. *SIGSOFT Softw. Eng. Notes*, 30:1–7, 2005.

[13] I. Segall, R. Tzoref-Brill, and E. Farchi. Using Binary Decision Diagrams for Combinatorial Test Design. In *Proc. 20th Intl. Symp. on Software Testing and Analysis (ISSTA'11)*, pages 254–264. ACM, 2011.

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