# Exploring Test Composition: Towards Reusability in Combinatorial Test Design

Anna Zamansky
University of Haifa
Haifa, Israel
Email: annazam@is.haifa.ac.il

Eitan Farchi
IBM Research
Haifa, Israel
Email: farchi@il.ibm.com

***Abstract*—Combinatorial test design (CTD) is an effective test planning technique that reveals faulty feature interaction in a given system. CTD takes a systematic approach to formally model the system to be tested, aiming to minimize the number of test cases while ensuring coverage of given conditions or interactions between parameters. Since the system model and its test space in real-life cases are usually enormous, the process of creation of new tests is very expensive. This naturally leads to the need for exploring ways in which reuse methodologies can be incorporated into CTD practices. In this paper, we extend the standard CTD framework to a reuse-oriented setting by incorporating the notion of test *composition*. This notion arises naturally in sequential testing scenarios, where the output of one test is used as the input of the next test. Based on the proposed framework, we propose a reuse-oriented reformulation for the CTD problem, with the *composability* of test plan being the main consideration.**

***Keywords–combinatorial test design; pairwise testing***

## I. INTRODUCTION

As software systems become increasingly complex, verifying their correctness is even more challenging. Formal verification approaches are highly sensitive to the size of complexity of 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 potentially enormous space of scenarios, configurations or conditions that is typically exponential in nature.

The process of test planning refers to the design and selection of tests out of a test space aiming at reducing the risk of bugs while minimizing redundancy of tests. *Combinatorial Test Design* (CTD) ([1], [2]) is an effective test planning technique, in which the space to be tested, called a *combinatorial model*, is represented by a set of parameters, their respective values and restrictions on the value combinations [3]. The approach of CTD can be applied at different phases and scopes of testing, including end-to-end and system-level testing and feature-, service- and application program interface-level testing.

The standard general formulation of the CTD problem consists of finding a small (and ideally minimal) set of test cases (a subset of the space to be tested), which ensures coverage of given conditions, or interactions between variables (such as pairs, three-way, etc.) The most common special case of the CTD problem is pairwise testing, in which the interaction of every pair of parameters must be covered. This is justified by experiments showing that a test set that covers all possible pairs of parameter values can typically detect between 50 to 75 percent of the bugs in a program [4].

The process of building combinatorial models for CTD is a laborious and error-prone task, which involves finding a set of parameters and values that define the test space and correctly identifying all valid value combinations. One potential pitfall 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 cobminations. Providing support for the test space definition process is a crucial factor in a successful application of CTD techniques to a wide range of testing domains. IBM's tool FoCuS [5] aims to automatically assist the tester in the solution of CTD problems by constructing the test space efficiently, while considerably reducing the risk of omissions. In [3] recurring patterns in CTD models were studied. These patterns capture cases that often repeat in different CTD models of different types and from different domains. Solutions for how to translate them into parameters, values and restrictions already exist, and thus can fascilitate *model reuse*.

In this paper, we address another type of reuse in CTD, namely *test reuse*. Addressing this problem is important, as the test space in real-life cases is usually enormous, creating new tests is very expensive. In this work-in-progress we explore the idea of *reuse-oriented* CTD based on the notion of *composition of test plans*. We propose a formal framework in which composition of test plans can be defined. This leads to a natural notion of the design space, which is the inductive closure of test plans under composition. Based on the proposed framework, we propose a reuse-oriented reformulation for the CTD problem, with the *composability* of test plan being the main consideration.

## II. THE CTD FRAMEWORK

CTD is a powerful technique for functional testing. The most well-known versions of this approach aim at testing all value combinations of a given size t of the system's parameters. This problem of finding a minimal set of tests that cover all combinations of size $t$ can be reduced to the mathematically equivalent problem of finding a covering array of strength $t$ [6].

In this paper, we take a more general approach along the lines of [7], where the CTD problem is considered as a subset selection problem. Below we provide a generalization of the basic definitions of the framework, in which we make a separation between the input and output parameters of a test plan. This will be instrumental in what follows for defining the notion of composition.

 ISBN: 978-1-61208-449-7 

We assume a finite set of system parameters $\mathbf{Par} = \{\mathbf{A}_1, \ldots, \mathbf{A}_n\}$. We treat a parameter as a finite set of its possible values. For any value $a \in \mathbf{A}_i$, we say that $a$ has type $\mathbf{A}_i$, denoted by $type(a) = \mathbf{A}_i$. Subsets of parameters are called p-collections and are denoted by $\mathcal{A}, \mathcal{B}, ...,$.
We start by defining the notion of an (unordered) test, which is basically a collection of values the combination of which needs to be tested:

**Definition 1:** Let $\mathcal{A} = \{\mathbf{A}_1, \ldots, \mathbf{A}_m\}$ be a p-collection.

1) A **selection** for $\mathcal{A}$ is an element $\mathcal{S} \in \mathbb{P}(\bigcup_1^m \mathbf{A}_i)$ such that for distinct $a, b \in \mathcal{S}$, $type(a) \neq type(b)$. Write $\mathsf{Selections}_{\mathcal{A}}$ for the set of all selections for $\mathcal{A}$.
2) Call a selection $\mathcal{S}$ for $\mathcal{A}$ an **unordered test** when for every $\mathbf{A} \in \mathcal{A}$, there is some $a \in \mathcal{S}$, such that $type(a) = \mathbf{A}$ (so that $|\mathcal{S}| = |\mathcal{A}|$). Write $\mathsf{Tests}_{\mathcal{A}}$ for the set of all unordered tests for $A$.

**Example 1:** $\mathcal{A} = \{\mathsf{FileOps}, \mathsf{PathName}, \mathsf{OS}\}$ be a p-collection (taken out of a larger superset of parameters), where

$$\mathsf{FileOps} = \{open, close, read, write\}$$

$$\mathsf{PathName} = \{relative, absolute\}$$

$$\mathsf{OS} = \{unix, windows\}$$

Then $\mathcal{S}_1 = \{open, relative, unix\}$ and $\mathcal{S}_2 = \{close, windows\}$ are selections for $\mathcal{A}$, while the former is also an unordered test.

Using sets of selections, we can easily capture various interaction modes, such as pairwise testing (as well as others):

**Example 2:** For a p-collection $\mathcal{A} = \{\mathbf{A_1}, \ldots, \mathbf{A_m}\}$, denote by $2Pair(\mathcal{A})$ the set of all selections for $\mathcal{A}$ of size 2.

Thus we think of tests just as sets, specifying which values interact with which. However, when actually running tests, other types of information become important. One of them is a separation between the *input* and *output* parameters, which defines possible orders of execution of tests as sequences of consecutive test runs. Another issue is the *order* of parameter appearance. Thus, when defining the notion of *executable test set*, i.e., those tests that can actually be run in the system, we incorporate the above types of information as well:

**Definition 2:** Let $\mathcal{A} = \{\mathbf{A}_1, \ldots, \mathbf{A}_m\}, \mathcal{B} = \{\mathbf{B}_1, \ldots, \mathbf{B}_k\}$ be p-collections.

- For $e = (a_1, \ldots, a_r) \in (\mathbf{A}_1 \times \ldots \times \mathbf{A}_m) \times (\mathbf{B}_1 \times \ldots \times \mathbf{B}_k)$ (where $r = m + k$), we define the collapse of $e$ by $col(e) = \{a_1, \ldots, a_r\}$.
- An element $e \in (\mathbf{A}_1 \times \ldots \times \mathbf{A}_m) \times (\mathbf{B}_1 \times \ldots \times \mathbf{B}_k)$ is called an **ordered test** if $col(e)$ is an unordered test.
- An executable test set from $\mathcal{A}$ to $\mathcal{B}$, denoted by $\mathcal{E} : \mathcal{A} \rightarrow B$, is a set of ordered tests.

**Definition 3:** Let $\mathcal{A}, \mathcal{B}, \mathcal{C}$ be p-collections.

- A **pre-test plan** from $\mathcal{A}$ to $\mathcal{B}$, denoted by $P : \mathcal{A} \rightarrow \mathcal{B}$ is a triple $P = (\mathcal{E}, \mathbf{R}, \mathbf{T})$ where:
  - $\mathcal{E} : \mathcal{A} \rightarrow \mathcal{B}$ is an executable test set.
  - $\mathbf{R}$ is a set of selections for $\mathcal{A} \cup \mathcal{B}$ called *coverage requirements*,
  - $\mathbf{T}$ is a set of selections for $\mathcal{A} \cup \mathcal{B}$, such that $\mathbf{T} \subseteq col(\mathcal{E})$,

  A **test plan** from $\mathcal{A}$ to $\mathcal{B}$ is a pre-test plan $P = (\mathcal{E}, \mathbf{R}, \mathbf{T})$ which satisfies that for every $\mathcal{R} \in \mathbf{R}$, there is some $\mathcal{T} \in \mathbf{T}$, such that $\mathcal{R} \subseteq \mathcal{T}$ (in words: every coverage requirement $\mathcal{R}$ in $\mathbf{R}$ is 'covered' by some test $\mathcal{T}$ in $\mathbf{T}$.)

**Example 3:** Let $\mathcal{A}$ be the p-collection from Example 1. Let $\mathcal{B} = \{\mathsf{PackageSize}, \mathsf{AckRequired}\}$ be a p-collection, where

$$\mathsf{PackageSize} = \{1KB, 2KB, 3KB\}$$

$$\mathsf{AckRequired} = \{yes, no\}$$

Let $\mathcal{C} = \{\mathsf{Mode}, \mathsf{Protocol}\}$ be a p-collection, where

$$\mathsf{Mode} = \{on, off\}$$

$$\mathsf{Protocol} = \{UDP, TCP\}$$

We can define, e.g., the following test plans $P_1 : \mathcal{A} \rightarrow \mathcal{B}$ and $P_2 : \mathcal{B} \rightarrow \mathcal{C}$: $P_1 = (\mathcal{E}_1, \mathbf{R}_1, \mathbf{T}_1)$, where:

$$\mathcal{E}_1 = (\mathsf{FileOps} \times \mathsf{PathName} \times \mathsf{OS}) \times (\mathsf{PackageSize} \times \mathsf{AckRequired})$$

$$\mathbf{R}_1 = \{\{relative, unix, yes\}\}$$

$$\mathbf{T}_1 = \{\{open, relative, unix, 2KB, yes\}\}$$

$P_2 = (\mathcal{E}_2, \mathbf{R}_2, \mathbf{T}_2)$, where:

$$\mathcal{E}_2 = (\mathsf{PackageSize} \times \mathsf{AckRequired}) \times (\mathsf{Mode} \times \mathsf{Protocol})$$

$$\mathbf{R}_2 = \{\{yes, TCP\}\}$$

$$\mathbf{T}_2 = \{\{3KB, yes, on, TCP\}\}$$

We are now ready to state the standard formulation of the CTD problem in terms of our framework: *given the set of executable tests $\mathcal{E} : \mathcal{A} \rightarrow \mathcal{B}$ and coverage requirements $\mathbf{R}$, find a set of tests $\mathbf{T}$, such that $P = (\mathcal{E}, \mathbf{R}, \mathbf{T})$ is a valid test plan.* In the case of pairwise testing, e.g., we set $\mathbf{R} = 2Pair(\mathcal{A} \cup \mathcal{B})$, where the latter is taken from Example 2.

Many algorithms and tools exist for solving various instances of the CTD problem ([8]). They can be classified into three categories ([9]):

- *algebraic*: providing a solution by a mathematical construction. These approaches usually lead to optimal results, but are however highly inefficient in practice (see, e.g., [10]).
- *greedy*: applying some search heuristic to incrementally build the solution. An efficient algorithm of such kind based on Binary Decision Diagrams [11] is given in [7], other examples are [12], [13].
- *meta-heuristic*: applying genetic or other bio-inspired search techniques (see, e.g., [14], [15]).

 ISBN: 978-1-61208-449-7

## III. Introducing Composition

The intuition behind composition of test plans is very simple. Suppose a user has designed tests plans for several parts of the system. Now, he may want to test several parts sequentially, running one after the other and using the output of one as input for the other. Instead of constructing tests from scratch, he can reuse existing tests for the system parts via *composition*. To define in precise terms the notion of composition of test plans, we start by defining a composition of each of their parts: executable test sets and selections (and so also of tests).

**Definition 4:** Let $\mathcal{A} = \{\mathbf{A}_1, \ldots, \mathbf{A}_m\}$ and $\mathcal{B} = \{\mathbf{B}_1, \ldots, \mathbf{B}_k\}$. Let $\mathcal{S}_A, \mathcal{S}_B$ be selections for $\mathcal{A}$ and $\mathcal{B}$ respectively. Let $\mathcal{E}_1 : \mathcal{A} \to \mathcal{B}$ and $\mathcal{E}_2 : \mathcal{B} \to \mathcal{C}$ be executable test plans.

- Define $\mathcal{E}_2 \circ \mathcal{E}_1 = \{(a, c) \mid \exists b \in \mathbf{B}_1 \times \ldots \times \mathbf{B}_k.(a, b) \in \mathcal{E}_1 \wedge (b, c) \in \mathcal{E}_2\}$.
- If for every $a \in \bigcup_{1 \le i \le m} \mathbf{A_i} \cup \bigcup_{1 \le i \le k} \mathbf{B_i}$ it holds that $type(a) \in \mathcal{A} \cap \mathcal{B}$ implies $a \in \mathcal{S}_A \cap \mathcal{S}_B$, we say that $\mathcal{S}_A \circ \mathcal{S}_B$ is well-defined and equals to $\mathcal{S}_A \cup \mathcal{S}_B \setminus \mathcal{S}_A \cap \mathcal{S}_B$.
- Define $S_A \circ S_B = \{\mathcal{S}_A \circ \mathcal{S}_B \mid \mathcal{S}_A \circ \mathcal{S}_B$ is well defined, $\mathcal{S}_A \in S_A, \mathcal{S}_B \in S_B\}$.

The above definition captures both compositions of coverage requirements and of tests (as both of them are sets of selections). Note also that composition of 2-pair coverage requirements leads to 2-pair coverage requirements:

**Proposition 1:** Let $\mathcal{A}, \mathcal{B}, \mathcal{C}$ be p-collections. Then

$$2Pair(\mathcal{A} \cup \mathcal{B}) \circ 2Pair(\mathcal{B} \cup \mathcal{C}) = 2Pair(\mathcal{A} \cup \mathcal{C})$$

We can now define the following natural notion of composition of test plans:

**Definition 5:** Let $P_1 = \langle \mathcal{E}_1, \mathbf{T_1}, \mathbf{R_1} \rangle : \mathcal{A} \to \mathcal{B}$ and $P_2 = \langle \mathcal{E}_2, \mathbf{T_2}, \mathbf{R_2} \rangle : \mathcal{B} \to \mathcal{C}$ be test plans. The composition of $P_1$ and $P_2$ is defined by the pre-test plan

$$P_2 \circ P_1 = \langle \mathcal{E}_2 \circ \mathcal{E}_1, \mathbf{T_2} \circ \mathbf{T_1}, \mathbf{R_2} \circ \mathbf{R_1} \rangle$$

To capture the set of all possible runs of sequences of tests for which the output of one test is used as the input of the next test, we can now define the notion of the design space, which contains pre-test plans:

**Definition 6:** Let $P_1 : \mathcal{A}_1 \to \mathcal{B}_1, \ldots, P_n : \mathcal{A}_n \to \mathcal{B}_n$ be test plans from $A_i$ to $B_i$ respectively. For $i \ge 0$, define the sets $\mathbf{DS}_i(P_1, \ldots, P_n)$ of pre-test plans as the smallest sets satisfying:

- $P_1, \ldots, P_n \in \mathbf{DS}_0(P_1, \ldots, P_n)$.
- If $P_i, P_j \in \mathbf{DS}_k(P_1, \ldots, P_n)$ and $\mathcal{B}_i = \mathcal{A}_j$, then $P_i \circ P_j \in \mathbf{DS}_{k+1}(P_1, \ldots, P_n)$.

We denote $\bigcup_{i \ge 0} \mathbf{DS}_i$ by $\mathbf{DS}$.

## IV. Reuse-Oriented Test Design

Given existing test plans, e.g., for $\mathcal{A} \to \mathcal{B}, \mathcal{B} \to \mathcal{C}, \mathcal{B} \to \mathcal{D}$, one can compose them to obtain new test plans for $\mathcal{A} \to \mathcal{C}$ and $\mathcal{A} \to \mathcal{D}$. The challenge is, however, dealing with the fact that composing valid test plans does not necessarily lead to valid test plans. Indeed, the operation of composition does not preserve validity of test plans, as demonstrated by the following example:

**Example 4:** Consider again the test plans $P_1$ and $P_2$ from Example 3. Their composition is the pre-test plan $P_2 \circ P_1 = (\mathcal{E}, \mathbf{T}, \mathbf{R})$, where:

$$\mathcal{E} = (\mathsf{FileOps} \times \mathsf{PathName} \times \mathsf{OS}) \times (\mathsf{Mode} \times \mathsf{Protocol})$$

$$\mathbf{T} = \emptyset$$

$$\mathbf{R} = \{\{relative, unix, TCP\}\}$$

This pre-test plan is obviously not a test plan, as there is no test meeting the requirement in $\mathbf{R}$.

We now come to the key idea of reuse-oriented CTD problem based on composition: designing basic tests with the aim of effectively composing them at later stages when required.

**Definition 7:** Two test plans $P_1 = \langle \mathcal{E}_1, \mathbf{T_1}, \mathbf{R_1} \rangle : \mathcal{A} \to \mathcal{B}$ and $P_2 = \langle \mathcal{E}_2, \mathbf{T_2}, \mathbf{R_2} \rangle : \mathcal{B} \to \mathcal{C}$ are *composable* if $P_2 \circ P_1$ is a valid test plan.

It is easy to see that for every pair of executable test sets $\mathcal{E}_1, \mathcal{E}_2$ and coverage requirements for $\mathcal{A} \to \mathcal{B}$ and $\mathcal{B} \to \mathcal{C}$ respectively, there always exist valid test plans $P_1 = \langle \mathcal{E}_1, \mathbf{T_1}, \mathbf{R_1} \rangle : \mathcal{A} \to \mathcal{B}$ and $P_2 = \langle \mathcal{E}_2, \mathbf{T_2}, \mathbf{R_2} \rangle : \mathcal{B} \to \mathcal{C}$, which are composable.

The notion of composability can be naturally extended to any number of test plans, as well as to the inductive closure under composition $\mathbf{DS}$ of a given depth:

**Definition 8:** Let $P_1 : \mathcal{A}_1 \to \mathcal{B}_1, \ldots, P_n : \mathcal{A}_n \to \mathcal{B}_n$ be test plans from $A_i$ to $B_i$ respectively. The design space $\mathbf{DS}$ is composable up to depth $k$ if for every $P \in \mathbf{DS}_i$ and $P' \in \mathbf{DS}_j$ for $i, j \le k$, such that $\mathcal{B}_i = \mathcal{A}_j$, $P_i \circ P_j$ is composable.

The notion of composability defined above gives rise to new interesting formulations of reuse-oriented versions of the CTD problem, such as the following: given executable test sets $\mathcal{E}_1, \mathcal{E}_2$ coverage requirements and $\mathbf{R_1}$ and $\mathbf{R_2}$ for $\mathcal{A} \to \mathcal{B}$ and $\mathcal{B} \to \mathcal{C}$ respectively, find $\mathbf{T_1}, \mathbf{T_2}$, such that $P_1 = \langle \mathcal{E}_1, \mathbf{T_1}, \mathbf{R_1} \rangle : \mathcal{A} \to \mathcal{B}$ and $P_2 = \langle \mathcal{E}_2, \mathbf{T_2}, \mathbf{R_2} \rangle : \mathcal{B} \to \mathcal{C}$ are *composable* test plans. This can also be extended to composability of a design space of a fixed depth in a natural way.

Similarly to the standard CTD problem, the above problems may be solved using appropriate algebraic, greedy or meta-heuristic algorithms.

## V. Summary and Further Work

In this paper, we define a formal framework, in which we formulate an algorithmic problem of reuse-oriented CTD based on composition. Composition of tests naturally arises in sequential testing scenarios, where the output of one test is

 ISBN: 978-1-61208-449-7

used as the input of the next test. An evaluation of our framework is the next natural step. It will be done by implementing a greedy algorithm for solving the composition-based CTD problem defined in this paper, by extending the algorithm of [7], based on Binary Decision Diagrams.

Natural operations on test plans other than composition can be further considered for a systematic test reuse (e.g., the standard $join$ operation from database theory). Another direction for future research is investigating which data structures are most suitable for providing efficient solutions to the type of algorithmic problems formulated in this paper.

## REFERENCES


[1] C. Nie and H. Leung, "A survey of combinatorial testing," ACM Computing Surveys (CSUR), vol. 43, no. 2, 2011, p. 11.

[2] J. Zhang, Z. Zhang, and F. Ma, "Introduction to combinatorial testing," in Automatic Generation of Combinatorial Test Data. Springer, 2014, pp. 1–16.

[3] I. Segall, R. Tzoref-Brill, and A. Zlotnick, "Common patterns in combinatorial models," in Proceedings of the IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST). IEEE, 2012, pp. 624–629.

[4] S. R. Dalal, A. Jain, N. Karunanithi, J. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, "Model-based testing in practice," in Proceedings of the 21st International Conference on Software engineering. ACM, 1999, pp. 285–294.

[5] http://researcher.watson.ibm.com/researcher/view_group.php?id=1871.

[6] A. Hartman and L. Raskin, "Problems and algorithms for covering arrays," Discrete Mathematics, vol. 284, no. 1, 2004, pp. 149–156.

[7] 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. ACM, 2011, pp. 254–264.

[8] M. Grindal, J. Offutt, and S. F. Andler, "Combination testing strategies: a survey," Software Testing, Verification and Reliability, vol. 15, no. 3, 2005, pp. 167–199.

[9] M. B. Cohen, M. B. Dwyer, and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," in Proceedings of the 2007 International Symposium on Software testing and analysis. ACM, 2007, pp. 129–139.

[10] N. Kobayashi, T. Tsuchiya, and T. Kikuno, "Non-specification-based approaches to logic testing for software," Information and Software Technology, vol. 44, no. 2, 2002, pp. 113–121.

[11] S. B. Akers, "Binary decision diagrams," IEEE Transactions on Computers, vol. 100, no. 6, 1978, pp. 509–516.

[12] 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, vol. 23, no. 7, 1997, pp. 437–444.

[13] K. Tai and Y. Lie, "A test generation strategy for pairwise testing," IEEE Transactions on Software Engineering, vol. 28, no. 1, 2002, pp. 109–111.

[14] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn, "Constructing test suites for interaction testing," in Proceedings of the IEEE 25th International Conference on Software Engineering, 2003, pp. 38–48.

[15] K. J. Nurmela, "Upper bounds for covering arrays by tabu search," Discrete applied mathematics, vol. 138, no. 1, 2004, pp. 143–152.


 ISBN: 978-1-61208-449-7 