2012 IEEE Fifth International Conference on Software Testing, Verification and Validation

Common Patterns in Combinatorial Models

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 systematically exercises interactions between parameters of the test space. 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.**

This paper describes patterns that we have found to be recurring in combinatorial models, i.e., recurring properties of the modeled test spaces. These patterns are often hard to identify and capture correctly in a model, thus are common pitfalls in combinatorial modeling. We describe these patterns, supply methods for identifying them, and suggest simple yet effective solutions for them.

I. Introduction

It is a well known fact that software has defects. While verification approaches for software, such as formal verification and model-based testing, do exist, they are highly sensitive to the size and complexity of the system. Functional testing is therefore a part of practically every software development lifecycle. However, clearly one can never execute all possible tests on a given system. Therefore, test planning is an important step for effective testing – choosing out of the almost endless list of possible tests which ones to actually execute.

Combinatorial test design (CTD), also referred to as combinatorial testing (CT), is an effective test planning technique that systematically exercises interactions between parameters of the test space. In CTD, the test space is modeled by a set of parameters, their respective values, and restrictions on the value combinations. We hereby refer to such models as *combinatorial models*. 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 instance 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 [9], [2]. Other experimental work has shown that typically 100% of bugs can be revealed by covering the interaction of between 4 and 6 parameters [5].

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 model that captures the valid test space is one of the main deployment obstacles. Typically, the parameters of a combinatorial model do not map directly to user inputs of the system, but are rather a high level representation of functional points of variability between tests. Thus, identifying correctly the set of parameters for the model, their values, and the restrictions between them may be a laborious task that requires skills such as abstract thinking and applying of different logics. We found out that less experienced CTD practitioners, even if they are highly experienced testers or test architects, tend to overlook significant aspects of combinatorial models that are crucial for effective application of CTD.

This paper describes recurring patterns in combinatorial models, i.e., recurring properties of the modeled test spaces, which are often hard to identify and capture correctly in a model. We encountered these patterns in many different models, regardless of the domain of the system under test, the current level of testing (e.g., system test, function test), etc. We describe these patterns, which are common pitfalls in combinatorial modeling, and supply simple yet effective solutions for them.

There are three categories of pitfalls in combinatorial models – correctness (a model that does not capture correctly what it is intended to capture), completeness (a model that omits an important part of the test space) and redundancy (a model that explicitly enumerates different cases that are actually equivalent). Each modeling pattern described in this paper addresses one or more of these categories. Each section below describes a single pattern, along with the categories to which it is related. Table I summarizes the patterns and the categories addressed by them.

Listing some of the most common patterns, along with the issues that they address, can help testers identify which

978-0-7695-4670-4/12 $26.00 © 2012 IEEE
DOI 10.1109/ICST.2012.150

Table I
SUMMARY OF THE MODELING PATTERNS PRESENTED IN THIS PAPER, AND THE PITFALL CATEGORIES THAT THEY ADDRESS

Pattern	Correctness	Completeness	Redundancy
Optional and conditionally-excluded values	+	+	
Multi-selection	+	+	
Ranges and boundaries		+	+
Order and padding		+	
Multiplicity and symmetry			+

common pitfalls they are facing, as well as suggest how to correctly capture the intended test space behavior in the combinatorial model. This list can be used as a "check list" when reviewing the different aspects of the model – correctness, completeness and redundancy. We also mention, for each pattern, questions that may be asked in order to identify when this pattern is applicable. We believe that by considering the patterns in this paper, one can significantly improve the quality and effectiveness of combinatorial models.

Note that by no means do we claim this to be a complete list of all modeling patterns relevant to combinatorial models. We describe here the patterns that we have found most useful in our experience.

II. OPTIONAL AND CONDITIONALLY-EXCLUDED VALUES

This pattern addresses two, similar, frequently occurring modeling mistakes that cause a model to be incomplete, or, in the case of modeling using Cartesian Products, even incorrect.

Consider the following example. An online form has two fields – an e-mail field that is mandatory (cannot be empty), and a home address field that is optional. A simple minded model would be:

```
  EMail - valid/invalid
  HomeAddr - valid/invalid
```

But this model is incomplete, because it does not distinguish between the cases of a valid home address and an empty field, that most probably follow different code paths. To cover both cases an extra attribute value has to be added:

```
  EMail - valid/invalid
  HomeAddr - valid/empty/invalid
```

Conditionally excluded values look similar, but may have a higher impact. In an online form case where there is an e-mail address, a home address, an indication that billing by e-mail is required, and a condition that the home address is used only if billing by e-mail is not required, the following model seems natural:

```
  EMail - valid/invalid
  HomeAddr - valid/invalid
  BillByEMail - true/false

NotAllowed:
  HomeAddr == ``Valid'' && BillByEmail
  HomeAddr == ``Invalid'' && BillByEmail
```

Natural as it seems, this model is incorrect – it does not allow any combination that requests billing by email! A correct and complete model for this case (addition highlighted in bold) is:

```
  EMail - valid/invalid
  HomeAddr - valid/empty/invalid/
              NotApplicable
  BillByEMail - true/false

NotAllowed:
  HomeAddr == ``Valid'' && BillByEmail
  HomeAddr == ``Empty'' && BillByEmail
  HomeAddr == ``Invalid'' && BillByEmail
  HomeAddr == ``NotApplicable'' &&
!BillByEmail
```

Note that the NotApplicable value has to be restricted to prevent its use where other attribute values are applicable. Omitting this restriction is another common pitfall.

The restrictions can be simplified to be:

```
NotAllowed:
  HomeAddr != ``NotApplicable'' &&
BillByEmail
  HomeAddr == ``NotApplicable'' &&
!BillByEmail
```

This two-way form is the typical restriction pattern for models that describe alternative paths.

The pattern of conditionally excluded values is not an issue in tree based modeling [3], where some attributes exist only on part of the branches. Of course, identifying these attributes and their values has to be done correctly.

III. MULTI-SELECTION

The Multi-Selection pattern addresses correctness and completeness, when a parameter should be allowed to get more than one value in a single test.

Traditionally, combinatorial models consist of parameters and values for them. A test in this setting is an assignment of exactly one value to each parameter. On the other hand, a very common situation is one in which a parameter may get more than one value in a single run. Consider, for example, a shopping cart system, in which one may have meat items, dairy ones, fish and drinks. The cart may include items of more than one type.

Inexperienced practitioners might overlook the multi-selection issue altogether, and model this system using a single parameter with four values:

```
Cart Item - meat/dairy/fish/drinks
```

Such a model clearly allows for only one type of item to be in the cart, and does not exercise interactions between the types at all. Alternatively, one could model this system by duplicating the above mentioned parameter t times, where t is the interaction coverage used (e.g., 2 in pairwise testing):

```
Item 1 - meat/dairy/fish/drinks
Item 2 - meat/dairy/fish/drinks/none
```

This is a common modeling decision, which allows one to test interactions between values of different kinds. However, in this example, the shopping cart is limited to 2 items. Moreover, the size of the test plan now has a lower bound on its size of n^t tests, where n is the number of values for this parameter, and t is the level of interaction, since all combinations for the t parameters should be covered.

Therefore, we suggest to model multi-selection parameters as n different Booleans, one for every value. In our example, the model would be:

```
Meat - true/false
Dairy - true/false
Fish - true/false
Drinks - true/false
```

By choosing this modeling pattern, one does not limit the number of items that a shopping cart may contain, yet still covers interactions between all values of the multi-selection parameter. Note that the non-appearance of a value from the original model is also represented in this solution explicitly by the `false` value (e.g., not having meat in the cart). One may take advantage of this in cases where this non-appearance may be of importance in tests.

One should take special care with the extreme case in which all parameters are false (an empty cart in our example). In some models, this case should be excluded using restrictions. In others, it should be treated as a negative, bad path, case. Finally, in some models, it may be perfectly valid positive test.

A more general example of the multi-selection pattern appears in Section VI.

IV. Ranges and Boundaries

The Ranges and Boundaries pattern addresses both redundancy and completeness, by choosing how to represent multiple cases of a single parameter. When these cases can be divided into ranges, this pattern is applied: each range represents cases that are equivalent to each other with respect to the testing scenario. The boundaries of the ranges represent corner cases that need to be captured by the model and tested explicitly, since bugs tend to hide in corner cases.

For example, consider an algorithm for processing files. The algorithm takes different decisions based on the size of the file. When the file size is up to 1 MB, it is handled by a first approach. When the file size is between 1 and 256 MB, it is handled by a second approach. When the file size is greater than 256 MB, it is handled by a third approach. A natural parameter in the combinatorial model for testing this algorithm would be the size of the file to be handled. The parameter values should represent all ranges and boundary values: `0`, `moreThan0LessThan1`, `1`, `moreThan1LessThan256`, `256`, `moreThan256`. The values 0, 1, and 256 are the boundary values, while the rest are values representing ranges.

In the example above, it was quite easy to detect the ranges and boundaries. The numeric nature of the file size and the description that divided it into ranges naturally suggest using this pattern. However, there are some cases where detecting ranges and boundaries or even identifying that such exist may be more challenging. Consider another example, where a system allows a user to update an insurance plan. The user chooses the benefits for the updated insurance plan, where some of the benefits may already exist in the current test plan. When creating a combinatorial model for testing this feature of the system, one should first identify that the question whether the benefits chosen by the user for the updated plan already exist in the current plan is a factor that can influence the outcome. Therefore, it should be represented as a parameter in the model. Furthermore, the different answers to this question can be represented as ranges and boundaries, by choosing values as follows: `allBenefitsExisting`, `someExistingSomeNew`, `allBenefitsNew`. The first and last values are the boundary values, and the second value is the range. In this example, the ranges and boundaries were not given explicitly, but rather had to be deduced from the context.

It is very important to recognize the ranges and the boundaries correctly. Missing a boundary case can result in omissions in the test plan, and failing to recognize a range can result in redundancy in the test plan. The ranges and boundaries pattern is highly related to the well-known equivalence partitioning and boundary value analysis testing techniques, which are forms of value abstraction. These testing techniques are very popular in software testing, and especially in interface testing. The ranges and boundaries modeling pattern can be considered as a form of integration of these testing techniques with the CTD testing technique.

V. Order and Padding

The Order and Padding pattern addresses completeness when a requirement defines an output that depends on two (or more) inputs. In such cases, a modeler should ask whether it is likely that bugs can surface if inputs are presented in different orders.

For example, consider an expenses reimbursement policy that states: “expenses will be covered only if both food and lodging expenses are submitted”. A simple minded model for this requirement is

```
Lodging - true/false
Food - true/false
```

This model would result in four tests, with all combinations of submission of food and lodging expenses. However, one of the following bugs would not be detected with these tests (bugs highlighted in bold):

```
  if (record.type == ``Food'' &&
pLodgingWasSubmitted)
   { process(record); }

  if (record.type == ``Lodgin'' &&
pFoodWasSubmitted)
   { process(record); }
```

– or –

```
  if (record.type == ``Foood'' &&
pLodgingWasSubmitted)
   { process(record); }

  if (record.type == ``Lodgin'' &&
pFoodWasSubmitted)
   { process(record); }
```

In the first code fragment, if food was first in the test deck, the expenses would not be processed because the typo prevents recognition of the lodging expense. In the second example, the typo prevents correct processing when lodging was claimed first. Discovering these bugs requires a specific order of the attribute values in the input.

To account for input order we add another attribute (addition highlighted in bold):

```
Lodging - true/false
Food - true/false
FoodBeforeLodging - true/false/
                    NotApplicable
```

A complementing issue to the above is that of padding. To continue our example, padding addresses implementations of the expense reimbursement policy that depend on the records for expenses and lodging being consecutive or not, and may also have bugs when one of these records is first or last in the input. This calls for extending the model with padding related parameters, (addition highlighted in bold):

```
Lodging - true/false
Food - true/false
FoodBeforeLodging - true/false/
                    NotApplicable
PaddingBefore - true/false/
                NotApplicable
PaddingBetween - true/false/
                 NotApplicable
PaddingAfter - true/false/
               NotApplicable
```

The NotApplicable value for PaddingBetween is required for the case in which only one of food and lodging is claimed. The NotApplicable value for PaddingBefore and PaddingAfter are not necessary if an input without any of expenses and lodging cannot occur.

It is easy to see that when the number of parameters on which output depends grows, the number of order and padding related parameters grows too, and it grows much faster. A modeler should be careful to examine the problem at hand and create parameters only for input orders that may cause a bug in the tested software and not for those that will not.

VI. Multiplicity and Symmetry

The Multiplicity and Symmetry pattern addresses redundancy, where multiple elements of the same type appear within a system. When their interactions with other elements are equivalent, but this equivalence is not captured by the model, it can lead to redundancy in the model and in the resulting test plan.

Consider for example a system that contains two hosts and two storage devices. Each host can be of type x86 or PowerPC, while each storage device can be of type enterprise storage, mid-range storage or low-end storage. A model for this system could be:

```
host1 - x86/PowerPC
host2 - x86/PowerPC
storage1 - Enterprise/MidRange/LowEnd
storage2 - Enterprise/MidRange/LowEnd
```

However, by modeling the system as such, and requiring an interaction level of at least 2, one in fact requires the test plan to contain combinations representing equivalent interactions. For example, testing the case where **host1** is x86 and storage1 is Enterprise is the same as testing the case where **host2** is x86 and storage1 is Enterprise. The reason is that host1 of type x86 and host2 of type x86 are instances of the same software, and thus interact with other elements (in this case an Enterprise storage) exactly in the same way. Similarly, testing the case where **storage1** is MidRange and host2 is PowerPC is the same as testing the case where **storage2** is MidRange and host2 is PowerPC, again since storage1 of type mid-range and storage2 of type mid-range are instances of the same software. We may also want to avoid testing internal interactions between two instances of the same type, in case they are also equivalent. For example, testing the case where **host1** is x86 and **host2** is PowerPC may be the same as testing the case where **host1** is PowerPC and **host2** is x86. However, since this equivalence is not captured by the model, CTD cannot distinguish all these equivalent cases, and may produce more test scenarios than are really necessary.

It is important to note that a solution where only one host and one storage device are represented in the model may be insufficient. Though the interactions of the two hosts and the two storage devices with other elements may be similar, there are parts of the system that can be sensitive to the number of instances of each type that are running. For example, many types of race conditions can appear only when more than one instance is running. In addition, system hardware is frequently given in advance, and utilizing all of it is a goal for testing.

A possible solution to overcome this redundancy is as follows: instead of representing the hosts (or storage devices) as two parameters in the model, we represent each of their common values as a separate parameter in the model. By applying the multi-selection pattern, described in Section III, we indicate how many instances of this type are participating in the test. The new model would look as follows:

```
numx86Running - 0/1/2
numPowerPCRunning - 0/1/2
numEnterpriseRunning - 0/1/2
numMidRangeRunning - 0/1/2
numLowEndRunning - 0/1/2
```

In addition, we need to restrict the number of host types (or storage device types) that are running to match the number of hosts (or storage devices) in the system. This can be done as follows:

```
NotAllowed:
numx86Running + numPowerPCRunning != 2
numEnterpriseRunning +
numMidRangeRunning + numLowEndRunning
!= 2
```

This solution removes the redundant interactions since the new model does not specify to which host each host type is assigned, nor to which storage device each device type is assigned. Note that this transformation becomes laborious and error-prone as the number of parameters and parameter values grows.

An alterative solution that requires much less modeling effort on the one hand, but support in the CTD algorithm on the other hand, is as follows: the tester designates the equivalent parameters in the model (for example, the two hosts, the two storage devices), and whether their internal interactions are also equivalent. From this designation, The CTD algorithm derives sets of equivalent value tuples. During the generation of the test plan, once a tuple from such a set is covered by the test plan, all other tuples in the set are no longer required to be covered. Note that this solution requires the CTD algorithm to be able to handle dynamically changing coverage requirements. [8] describes a CTD algorithm that supports such requirements.

When a model contains multiple instances of the same type, this can hint that the multiplicity and symmetry pattern may apply. However, the instances do not have to be identical in order for the pattern to occur. Consider for example a case where the model contains two operating systems, one is basic and the other is upgraded. The upgraded system contains additional options that the basic one does not. Thus, the parameter that represents the upgraded system may contain additional values besides those of the basic system. However, the two operating systems may still be identical with respect to their common values, and thus their interactions with other elements may still be equivalent when they take one of those common values. In general, the multiplicity and symmetry pattern can apply also to elements that are not instances of the same type, but rather different types that are symmetric with respect to the testing scenario, though we have yet to encounter such a case in practice.

VII. Related Work

To the best of our knowledge, our work is a first attempt to systematically identify, describe and classify common recurring patterns in combinatorial models. However, modeling patterns in combinatorial models have been occasionally encountered and reported before (though not defined as modeling patterns), as part of a body of work on the application of CTD in practice. A recent survey on combinatorial testing [7] classifies 17 research papers as exploring the application of CTD to different levels of testing and types of systems, and 5 research papers exploring how to identify parameters, values, and interrelations between the parameters.

For example, the notion of optional values is mentioned in [6]. A special construct is described on how to support optional groups of values, which are a special case of the optional and conditionally excluded values pattern. [6] also describes a pattern related to redundant interactions, where interaction coverage is not required within a specific set of values for parameters, but is needed with other parameters. Note that this is a different case than the multiplicity and symmetry pattern that we describe, since multiplicity and symmetry still require the coverage of at least one representative from each set of equivalent interactions.

Recurring properties of test spaces that have been mentioned before but do not appear on our list of patterns are commonalities and hierarchies. Commonalities refer to the fact that parts of a specific test space can be common with other test spaces, and therefore can be reused. They are referred to in [6] as auxiliary aggregates. Hierarchies [1], [4] refer to the structure of the test space, where a sub-space can be defined and linked to a specific part of the test space, as a form of abstraction. While we do find these patterns useful, we have yet to use them in practice and therefore did not report them in this work.

The conclusion drawn in [7] on the research of the application of CTD in practice is that though many prior studies validate the fact that CTD can be applied to many types of systems, each research team followed its own testing procedure, and that better testing results can be obtained by following a more effective testing procedure. We believe that capturing common recurring patterns in combinatorial models can contribute to such a general and more effective testing procedure.

VIII. Summary and Future Work

In this paper, we describe five common patterns that occur in combinatorial models. We illustrate each pattern using one or more examples, supply methods for identifying the pattern, and suggest simple yet effective solutions for correctly capturing it in the model. We also indicate for each pattern which common pitfalls it addresses. The list of patterns, as well as the pitfalls addressed by them are summarized in Table I.

We plan to further extend the list of common patterns as we broaden our experience in applying CTD, and identify additional patterns from an in depth study of existing work on CTD modeling. In addition, we plan to add tool support for modeling patterns, both to help suggest patterns in given models and to take advantage of the detection of patterns to reduce the amount of manual modeling work that is required.

References

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

[2] 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.

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

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

[5] 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.

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

[7] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43(2):11:1–11:29, 2011.

[8] 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.

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