Proactive and Pervasive Combinatorial Testing

Dale Blue
IBM, USA

Orna Raz
IBM Research, Israel

Rachel Tzoref-Brill
IBM Research, Israel

Paul Wojciak
IBM, USA

Marcel Zalmanovici
IBM Research, Israel

ABSTRACT

Combinatorial testing (CT) is a well-known technique for improving the quality of test plans while reducing testing costs. Traditionally, CT is used by testers at testing phase to design a test plan based on a manual definition of the test space. In this work, we extend the traditional use of CT to other parts of the development life cycle. We use CT at early design phase to improve design quality. We also use CT after test cases have been created and executed, in order to find gaps between design and test. For the latter use case we deploy a novel technique for a semi-automated definition of the test space, which significantly reduces the effort associated with manual test space definition. We report on our practical experience in applying CT for these use cases to three large and heavily deployed industrial products. We demonstrate the value gained from extending the use of CT by (1) discovering latent design flaws with high potential impact, and (2) correlating CT-uncovered gaps between design and test with field reported problems.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Combinatorial Testing, Design Review, Desk Checking

ACM Reference Format:
Dale Blue, Orna Raz, Rachel Tzoref-Brill, Paul Wojciak, and Marcel Zalmanovici. 2018. Proactive and Pervasive Combinatorial Testing. In *ICSE-SEIP '18: 40th International Conference on Software Engineering: Software Engineering in Practice Track, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3183519.3183522

1 INTRODUCTION

As software systems are becoming more complex and release cycles are becoming shorter, there is a growing need to improve software quality while reducing testing time and cost. One of the effective techniques for coping with this challenge is combinatorial testing (CT) [3, 5, 6, 8, 12, 17, 21, 22]. The reasoning for CT is based on empirical data involving the analysis of thousands of software defects. The analysis showed that the defects appearance usually depends on the interaction between a small number of features of the system under test [1, 2, 13, 14, 20, 24]. This observation provided a new notion of coverage called *interaction coverage*, where the aim is to cover all the ways in which a small number of features (for example every 2 features) can interact. As opposed to traditional coverage which is measured on a test plan in a postmortem fashion, CT achieves full interaction coverage by construction of the test plan, as follows. One is required to manually model the features of the test space in the form of parameters, their respective values, and constraints on the value combinations (a.k.a a *combinatorial model*). A valid test in the test space is defined to be an assignment of one value to each parameter that satisfies the constraints. A CT algorithm automatically constructs a subset of the valid test space so that it covers all valid value combinations of every t parameters, where t is a user input.

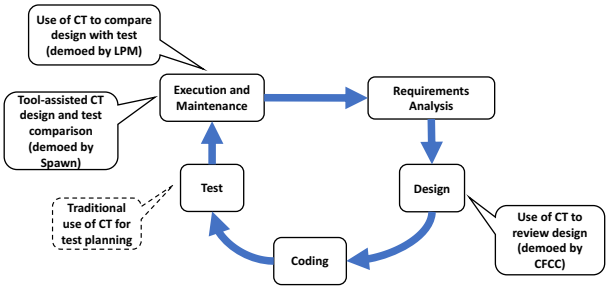

Figure 1: A single iteration of software development augmented with traditional use of CT (dashed line) and our novel use cases for CT (solid line). Note that the use of CT is indifferent to the development model (e.g., agile, continuous integration, spiral, etc.).

In this work, we extend the traditional use of CT to other parts of the development life cycle beyond testing and for other purposes beyond designing test plans. We further present three case studies – CFCC, LPM and Spawn, each capturing a different novel aspect of the usage of CT. Figure 1 displays a single iteration of software development, augmented with our suggested use cases. Whereas traditional use of CT occurs at test time to design a test plan (as depicted in the dashed line), we suggest to apply CT also at design time to improve design quality (bottom right side) and at execution and maintenance time to find gaps between design and test (top left side).

When applying CT at design phase, we build the combinatorial model based on design artifacts, and use it to generate scenarios for

ICSE-SEIP '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5659-6/18/05…$15.00
https://doi.org/10.1145/3183519.3183522

review of design and code, rather than to generate a test plan. We argue that the benefit from using CT for this purpose is twofold. First, design flaws may be found already during the application of the structured modeling process, i.e., while defining the design space in the form of a combinatorial model. Second, usage of scenarios to walk through the code or the design is a common review methodology, both for sequential code or design (a.k.a. 'desk checking') and for concurrent code or design (a.k.a. 'interleaving review technique') [9]. The reasoning for using CT to generate the list of scenarios for review rather than manually planning them is the same as the reasoning for using a test plan resulting from CT rather than a manually designed test plan. The list of scenarios resulting from CT is (1) of high quality since it achieves full interaction coverage, hence it is empirically likely to find most bugs, and (2) compact and hence feasible to use for a manual review process which requires human investment. CT achieves compactness by generating scenarios that are highly different from each other, maximizing the added value of each scenario in the list.

We demonstrate this use case on the structure management of Coupling Facility Control Code (CFCC), which is software that stores, manages, and secures shared data from high availability hosts. Since design flaws are found much earlier in the development life cycle, the potential savings is huge [11], and even more so in heavily deployed systems such as CFCC. Latent design flaws may surface due to different factors, for example added features, new hardware, or changes in usage patterns. Considering the large user base, the potential impact of each design flaw as well as the complexity of fixing it in such systems are even higher than in the case of early stage software.

When applying CT at execution and maintenance phase, we combine design phase modeling with test phase modeling by separately creating two combinatorial models, one based on design artifacts and the other based on test-related artifacts. We then compare the models in order to find gaps between design and test. Such a comparison can be effectively applied both to early stage software and to mature software that has legacy tests, and regardless of whether or not CT has been used to plan the original tests or review the original design.

We applied this use case to Live Partition Mobility (LPM), a facility for migration of running operating systems and their applications from one server to another server. An analysis of the gaps that were revealed as a result of the modeling effort indicates correlation with field problems. These problems as well as additional test plan gaps could have been avoided if combinatorial modeling were to be applied at an early design phase, both for review purposes and for test plan generation purposes.

The benefit in our suggested use cases is in being proactive and finding latent design and test problems before the users find them. A problem experienced by the user has extremely high costs in many domains. All of our case studies provide examples of such domains.

Lastly, we demonstrate that our cluster-based test plan functional analysis technology [23] can provide automated assistance in performing the second use case above, and hence reduce the effort and cost associated with manual modeling based on test-related artifacts. Specifically, our technology automatically and iteratively suggests model parameters and values when given an existing textual test plan as input. We demonstrate this on the Spawn functionality that is part of UNIX System Services of a high availability operating system. This programming interface provides a robust set of options that must work in many complex operating system states and environments. The model that is semi-automatically extracted from the tests is validated to be useful in that it maps well to the model that was manually created by a Subject Matter Expert (SME) based on design artifacts. In [23], we showed that this technique significantly reduces the human effort required for modeling when applicable, as is the case here.

For the combinatorial modeling and scenario generation we use the industrial-strength commercial CT tool IBM Functional Coverage Unified Solution (IBM FOCUS) [10, 19].

To summarize, our contributions are as follows: extend the use of CT to design phase in order to find latent design flaws, and to execution and maintenance phase in order to find gaps between design and test; applications of these novel use cases to real-world industrial products while demonstrating the value gain; and the use of a recently suggested novel technology to reduce the effort and increase the value of the latter use case.

The next section describes the methodology and technology we use for applying CT at design and to compare design with test. Section 3 presents the results from applying our methodology and technology to three real-world case studies. Section 5 discusses related work. In Section 4 we discuss lessons learned and value identified from our experience, and Section 6 concludes.

2 MODELING METHODOLOGY AND TECHNOLOGY

In this section, we describe our methodology and technology for applying CT for finding design flaws. We start with a general description of the CT application process, followed by our specific adjustments and enhancements to fit with our proposed novel use cases.

In [22], the CT methodology was described in detail, focusing on its application to system test. In the following, we provide relevant extracts from [22] that serve also for the current use cases.

The first step in applying CT is to define the test space in the form of a combinatorial model, consisting of parameters, their respective values, and constraints on value combinations. Constraints are rules defining which combinations are included in the test space and which are excluded from it. In the vast majority of cases, defining the test space is done manually by the tester. In [22], five distinct steps were defined: (1) identifying parameters, (2) assigning values to the parameters, (3) modeling bad path, (4) defining constraints on value combinations, and (5) defining interaction coverage requirements. When the test space is a functional test space, parameters are relatively easy to identify, as they conform with inputs to the function. In system test, identifying parameters might become trickier, hence [22] provides several suggestions for sources of parameters, such as flows with distinct steps, categories of system states, meaningful configuration characteristics, and orthogonal system operations or applications.

When assigning values to the parameters, well-known testing techniques such as equivalence partitioning and boundary value

analysis [15] are commonly used to help reach the right level of abstraction and avoid redundant testing.

Special consideration needs to be given to bad path modeling to guarantee correct and safe error handling. It is often the case that design flaws are revealed simply by systematic identification of potential error conditions, even before complete scenarios for review have been generated. Our CT tool supports explicit bad path modeling by allowing the user to indicate which value combinations represent error conditions, and automatically generating separate good path and bad path scenarios.

Defining constraints on value combinations is required in order to guarantee that the CT tool will generate only valid scenarios, i.e., scenarios that can be materialized into review scenarios or into executable test cases. An added value, similarly to bad path modeling, is that during this definition process, design flaws may often already be revealed, since the user is forced to think about combinations of conditions which may have not been considered when the design was conceived. Our CT tool supports several ways to locate and express constraints, including viewing different projections of the test space on subsets of parameters, and either excluding value combinations directly from the projection [19], or defining if-then rules that capture their intended relations.

Finally, another manual step is to define the interaction coverage requirements, i.e., the *interaction level* t so that all value combinations of size t must appear in the output plan. Naturally, the higher the value of t is, the bigger the size of the resulting plan will be. Some CT tools including ours support *variable strength generation* [4], where different interaction levels can be assigned to different subsets of parameters, to allow flexibility and incorporation of domain expert knowledge on parts of the modeled space that require extra attention. Interaction coverage requirements need not be determined a-priori, but rather since in most cases the tool automatically generates the output plan in a matter of seconds, different requirements can be experimented with until reaching an output plan which is both affordable and of sufficient quality in terms of the achieved interaction coverage.

The above process is traditionally applied by testers to define test spaces and generate test plans from them. When coming to apply CT at design phase to generate review scenarios, an important question is whether any changes need to be made in this process, and whether any differences exist between CT modeling for design and CT modeling for test. From our experience, while the technology and the overall process are similar in both cases, there are three significant differences, which we depict in Figure 2.

The first obvious difference is in the artifacts taken as input to the modeling process. Traditionally, a tester uses the system's external requirements to define the test space in the form of a combinatorial model, whereas the designer will use the design of the system. The second difference has to do with the modeling semantics and mind set. While a CT model for testing assumes that the design is sound and captures *what* the system under test is supposed to do with respect to its external requirements, when using CT to verify the design, the designer must question the design. This leads to a different type of models than the ones that capture test spaces, i.e., models that have to do with *how* the system is implemented. Design translates external requirements to design requirements. It removes

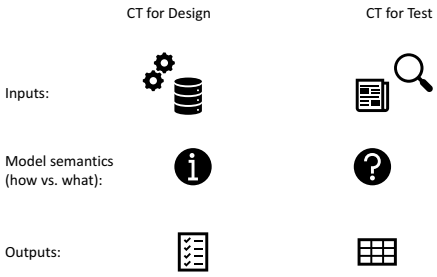

Figure 2: On the right, CT modeling for test plan generation. On the left, CT modeling for design review. Differences exist in (1) the input artifacts: requirements and test-related artifacts vs. design artifacts, (2) the modeling semantics: what is the system doing vs. how is it implemented, and (3) the output artifacts: a test plan vs. a list of scenarios for review.

ambiguity from the external requirements by making specific implementation choices that meet those requirements. When modeling the design space, the designer asks completely different questions, ones that have to do with what makes a good design. For example, are the design parts laid out in the best way? Are there any parts implemented in a way that might significantly increase coding and testing effort? Is there any ambiguity in the design which might lead to problems in coding?. These questions will lead to defining parameters and values that capture design requirements rather than external requirements, i.e., how the system is implemented rather than what it is intended to do. Given these differences in the type of models resulting from design phase modeling and test phase modeling, a natural question would be how can we compare them to each other, as we suggest in our second use case for applying CT at execution and maintenance phase. We will address this question shortly, when describing this use case.

The third difference is in the role of the CT output plan. Traditionally, CT is used for testing, i.e, for designing a test plan which is then translated into executable tests. In the novel use case we are suggesting, CT is used primarily as a review tool. The output of CT is a review plan, where every assignment of values to all parameters defines a review scenario. The review plan is used to perform the powerful desk checking review technique against the design. The desk checking technique directs the review to follow specific functionality and its inputs. This functionality may be implemented across multiple source code methods, as is often the case in object oriented code, for example. Desk checking is similar to executing code under a debugger, but may be executed at various levels of abstractions. This can be done even if no implementation exists yet, hence design flaws can be found much earlier and fixed much more easily. In addition, wasteful erroneous implementation of the wrong design is avoided.

Furthermore, when used at early design stage, the CT model and review plan provide a forward looking view in the sense that they help the designer estimate the coding and testing efforts, and plan accordingly. Beyond detecting design flaws, the designer can

use the model to understand at a very early stage how to simplify the design to reduce coding and testing effort. One example is to eliminate parameters when possible.

In our second case, when comparing design with test at execution and maintenance phase, we create two combinatorial models, one based on design artifacts and one based on test artifacts. The two models are then compared and mapped to each other in order to reveal design or test gaps. As we mentioned above, the two models will differ in the type of requirements they address and aspects they cover (external requirements for what the system is doing vs. design requirements for how it is implemented). This imposes challenges for comparison between them, such as different terminology used to express each of these two aspects. However the mapping between the two models still exists: each part of the test model capturing a certain external requirement should be mapped to parts of the design model capturing the design requirements for implementing that external requirement. A part of the test model that cannot be mapped to the design model may indicate gaps in the design or redundant testing. Similarly, a part of the design model that cannot be mapped to the test model may indicate gaps in testing or wrong/redundant design. Translating textual free form requirements, design, and test artifacts that may contain ambiguity to structured, concise and rigorous combinatorial models in the form of parameters and their values greatly assists in translating the terminology and creating the required mapping. Furthermore, as we will describe in Section 3, in our LPM and Spawn case studies we were able to overcome the terminology challenge and achieve successful mapping of the two models to each other.

Comparison of artifacts, each capturing a different aspect of the same system, is yet another common and useful review technique. In this case, the design flaws are found in a system which is already heavily deployed, and the savings is in being proactive and finding the latent design flaws before the user finds them. This can be done at any stage, and regardless of whether or not CT was used for the original design and testing.

As noted above, the actual generation of the output plan, be it a review plan or a test plan, is performed automatically by the CT tool. Hence, most of the effort in applying CT is in the modeling described above, which is usually an almost completely manual process.

To reduce the manual effort associated with modeling based on test artifacts, we use our cluster-based test plan functional analysis technology [23]. This technology receives legacy test cases as input, and then makes automated suggestions to the tester for combinatorial model parameters and values. In a nutshell, it performs text analysis of each test case in order to identify keywords and phrases. Clusters of test cases that make use of these keywords and phrases are then identified using the DBScan clustering algorithm [7], which does not require determining a-priori the number of clusters. The clusters are presented to the tester ordered by number of tests in the cluster and a numerical value termed *homogeneity* to rate how closely the tests in that cluster are correlated. Finally, as the tester chooses a cluster, she is led through a selection of the keywords and phrases in that cluster. These capture variability inside similar tests, and hence potentially stand for parameters and their values. This process is repeated for multiple clusters, and the resulting models are then combined into a single CT model. In [23], we defined a lower bound equation for the time savings achieved, based on the major contributing factors to the savings – the number, size, and homogeneity of the test clusters. We further reported on empirically measured time savings of over 80% in the creation of CT models using our technology, which exceeded our lower bound computation. In this work, we demonstrate that the model resulting from the use of our technique maps well to a model manually created by a domain expert. This provides further evidence for the effectiveness of the cluster-based approach in general, and specifically for reducing costs of comparison between models created based on test artifacts to those created based on design artifacts.

3 EVALUATION

We describe the case studies to which we applied the methodology described in Section 2. For the Coupling Facility Control Code (Section 3.1), we applied CT at design time in order to reveal design issues. For Live Partition Mobility (Section 3.2) and Spawn (Section 3.3), we applied CT to reveal gaps between design and test. Spawn tests were used to demonstrate the cluster-based functional analysis applicability in creating models from existing tests. For each case study we provide a description of the system, the use case we followed and our conclusions and lessons learned.

3.1 CFCC: The Coupling Facility Control Code

3.1.1 System description. The Coupling Facility (CF) stores, manages, and secures shared data from a high availability host operating system (OS) software and applications. The Coupling Facility Control Code (CFCC) is over 750,000 lines of code that perform the shared data management operations in the CF. The data residing in the CF consists of Lock, List, and Cache structures. As an example, OS database software is an exploiter of CF structure capabilities. By placing data in the CF, every OS member of a database group within a host shares the same data. This provides availability and performance advantages. CFCC has a central role to play by managing CF data structures within the host. Particular CFCC software optimization changes are the object for this part of our study. Our goal was to demonstrate that applying combinatorial modeling at the design phase for the purpose of generating scenarios for review can help reveal design issues and ultimately improve these CFCC optimization changes.

3.1.2 Methodology and results. A CT model for CFCC was created and used for design review, using the desk-checking review technique. The CFCC model was of modest size with 10 parameters. Around 10 hours cumulatively were invested by the CTD modeling expert and the CFCC designer. The review was performed on the 19 CT-generated scenarios and resulted in the identification and correction of two design flaws. The first issue involved an interaction that was not being properly handled by the code. This situation, if not addressed, would have resulted in problems for users deploying the enhanced CFCC software. Learning this, the team implemented a partial redesign with associated code changes. The resulting code changes were then reviewed again using the set of scenarios from the model. The experts were then convinced that the design and code enhancements passed all scenario requirements.

The second issue identified in the analysis revealed a combination of states where the code could lose track of structure consistency. This certainly would have been detected in function testing. Given that this issue was instead identified prior to code completion and any testing, the savings reflect the ideal result from doing combinatorial modeling at design time.

The feedback received from the CFCC design team indicated that CT was very good at producing scenarios to cover all the different states. There is some doubt whether even the best subject matter expert familiar with the data structure management design would be able to foresee all the state combinations to be checked. There would be no chance for a relative newcomer to the software team to identify all the state combinations.

3.1.3 Value. Combinatorial modeling for state coverage provided effective scenarios for review, simplifying the design evaluation task even for the subject matter expert. Using CT scenarios for design review found two design flaws in CFCC code. As design flaws are both extremely expensive when they manifest in a heavily deployed product and hard to find, finding even a single design flaw is a significant saving. CFCC state management (software locks, queues, multi-threading) is complicated and challenges even the best subject matter expert. CFCC kernel changes would be needed for the design change chosen for the modeling exercise. Such changes had proven expensive to test in terms of total test and debug hours invested.

There were no other design flaws found during testing or after release. Historical experience with CFCC kernel defect discovery, debug, fix creation and fix verification suggests that each ticket costs around 25 hours of cumulative developer and tester time. Given the elimination of two key defects by CT modeling, our estimate is a savings ratio of 1:5 regarding the time investment to apply CT in the manner described to fix the referenced CFCC flaws versus finding, diagnosing, and fixing those flaws during testing. The value and savings in finding the design flaws early significantly overcome the modeling cost.

3.2 LPM: Live Partition Mobility

3.2.1 System description. Business demands require that information technology (IT) service providers have hosted applications available around the clock. Customers want always on availability, despite the provider's need to periodically schedule system maintenance for software and hardware updates. On high availability servers, Live Partition Mobility (LPM) provides a means for migrating running operating systems and their applications from one server to another server. The migration is transparent to the application end users and the customer need not know their host was migrated across servers. LPM offers IT service providers the means to conduct system maintenance without impacting their users.

High availability servers support multiple operating systems with tens of thousands of applications. All this may be running simultaneously across hundreds of partitions. Thus, migration management software needs to account for compatibility, capability and readiness of every host to be migrated. Partition resources of processor, memory and networks all need to be factored into each migration. Given the scope of factors bearing on LPM success, the testing presents a reasonable challenge for CT. Originally, CT was not used to develop the LPM test plans.

3.2.2 Methodology and results. As part of this study, combinatorial models were developed to reflect LPM test coverage according to the test plan and according to design, and were then compared to each other. The motivation was to find both test gaps that correlate with field defects and design gaps. In the following we provide a detailed analysis and classification of the gaps found when comparing the LPM combinatorial models derived from design and from test.

(1) **Test plan gaps related to workload.** Explicit references to workload composition were largely absent from the LPM test plans, choosing instead to rely on implicit use of workloads common across the test organizations to get coverage. However, workload relevance is mentioned and implied throughout the design. As a consequence, the `client workload` model parameter was present in the design derived model yet missing from the test plans model. In all, six workload related parameters were derived from the design but not specifically in the test plans or test environments, two of which were associated with field problems. One thing that became apparent is that the design lists requirements describing internal code states or that represent timing related conditions consequent from different workloads running on the migrating clients. These states are not exposed to the tester. Testing this state coverage at a system level will require randomized workloads in addition to deterministic test set ups. In order to verify exhaustive coverage of these code states, the team is working on tracing and path flow analysis capability.
(2) **Gaps related to LPM interactions.** The `client migration interaction` parameter arose from the test plans, however, the design cites some specific examples of interactions from code outside LPM. The test plans list examples of non-LPM code functions that may interact with LPM. These functions are the values for the `client migration interaction` parameter. The design does not provide a broad treatment of possible function interactions from outside LPM code. Rather, the design addresses possible code interactions that the designers happened to think of, given their knowledge of the design. Interesting is that for the list of possible interactions mentioned in the design, none are cited in the test plan `client migration interaction` parameter values. Further, none of the interactions cited by the testers were noted in the design. The notion of non-LPM code functions interacting with LPM code seems to get a very different view point depending on whether the analyst is a system designer or a test designer. One of the interactions missing from the LPM test plan coverage correlated with a field problem.
(3) **Configuration and environmental test plan gaps.** Besides workload and non-LPM function interaction coverage, there were six additional parameters identified from the design review that reflected gaps in test plan coverage. These were related to configuration and environmental settings for the migrating client partition. Evaluation of these gaps led to their inclusion in the model for future tests.

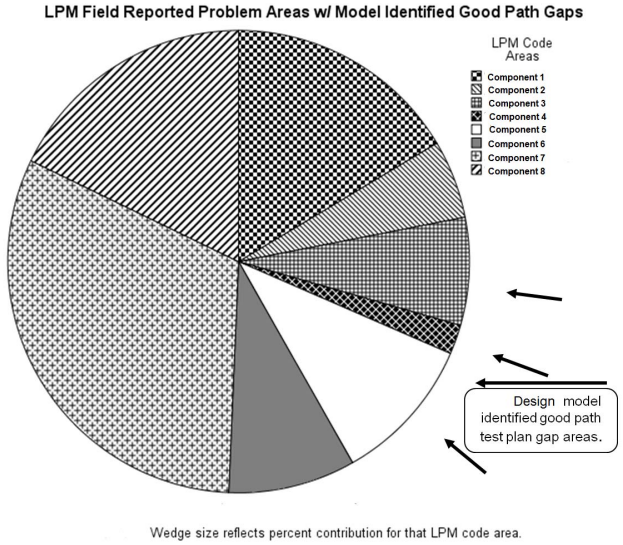

Figure 3: Model identified good path gaps in LPM field reported problem areas

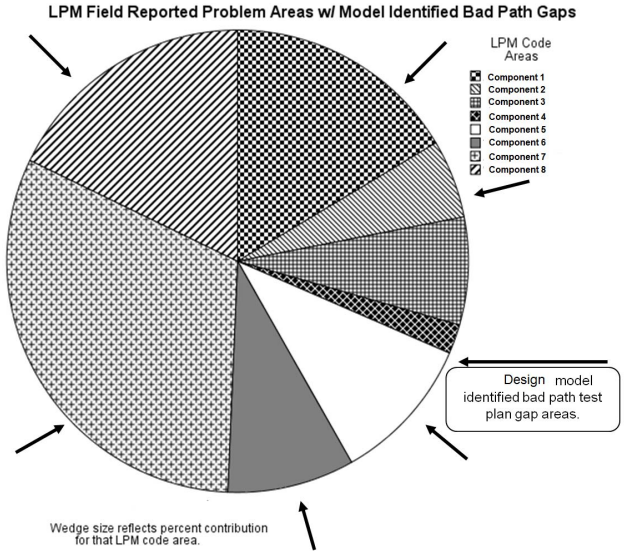

Figure 4: Model identified bad path gaps in LPM field reported problem areas

(4) **Bad Path test plan gaps.** The design review uncovered a number of bad path value combinations that were absent from the test plan. The corresponding tests would be either tests where the LPM operation is blocked from proceeding or tests where the LPM operation experiences a failing condition in the middle of the migration. The test plans had extensive coverage for the LPM failure handing hierarchy. This hierarchy results in a migration blocked from proceeding, a migration that backs out returning to the original state, or a failed migration that requires manual intervention. The gaps found were related to specific failure scenarios that would lead to one of those outcomes. In some cases the design-identified bad path scenarios did correlate with field problems.
(5) **Gaps due to terminology differences.** There was a challenge in mapping design parameters to test parameters due to the different terminology used to describe each of them. This is a direct result of the different levels of abstraction used to describe design versus test plan documents. For example, the design discusses migrating client capability and compatibility requirements. These properties are reflected in the test plans through parameters that relate to source and destination platform, software, and firmware versions.

Figures 3 and 4 depict the relative contribution to LPM field problems from each code area, and which of those areas intersect with design model identified good path and bad path test plan gaps, respectively. The design-based model identified good path gaps in 3 out of 8 code areas. It identified bad path gaps in 6 out of 8 code areas. Notably, considering good path and bad path collectively, test plan gaps were identified correlating with all field reported LPM code areas.

3.2.3 Value. LPM field problem types were aligned with configuration, documentation, or code defects. This last type, code defects, bears directly on testing. From a quantitative perspective, we examined what percentage of code defects intersected with test plan gaps uncovered with our combinatorial modeling. Our analysis revealed that these amounted to 20% of the code defects. This represents a significant percentage of field reported problems that were in fact correlated with test gaps identified through the design-based combinatorial model.

In addition, design gaps relating to LPM interactions were uncovered through the modeling process. Fortunately, this later point does not concern LPM customers since the software has already been tested and hardened for these interactions. Including these interaction references in the design closes these gaps. Taken collectively, this serves as considerable evidence of the combinatorial design modeling process value.

Furthermore, it was possible to map and compare the design based model with the test based model, providing evidence for the practicality of our approach of finding design and test gaps by comparing CT models.

As CT modeling was done after the fact we could compare the actual project and a hypothetical project involving CT. As mentioned above, about 20% of post release defects would have been eliminated prior to release by running the CT designed tests. Comparing the time to apply CT in the LPM project planning versus the way that LPM planning was conducted, we concluded they would have been about the same. Therefore, for the same cost we get a much bigger benefit: a significant reduction of post release issues for no appreciable test planning cost increase.

3.3 Spawn: Unix Spawn a Process

3.3.1 System description. The Unix System Services (USS) element of a high availability IBM operating system is a key element of its open and distributed computing platform. As a part of USS both the Spawn function and Spawn callable service combine the semantics of the fork and exec services to create a child process to run a specified operating system UNIX executable file. Concerns in testing the Spawn function include not only the input parameters but also many variables related to the system environment it is invoked in.

Evolution of the Spawn function provides an example of the challenges presented in maintaining legacy code. Test scenarios have been developed over years, as the function is impacted by changes in other parts of the underlying operation system components, the discovery of bugs, and the need to enhance the function with new capabilities. With each change, new tests are added resulting in a growth of the full test plan. Additionally, changes in who maintains the code over time add a loss of understanding of the path coverage of the test plan.

Our goal was to demonstrate that tool assistance is possible and effective in creating test based models for the use case of finding gaps between design with test .

3.3.2 Methodology and results. The Spawn case study produced two models: a manual model created by a SME of the Spawn design space and a semi-automatic model created by a tester using the cluster-based test plan functional analysis technology [23] on existing Spawn tests. Once two different CT models exist, it becomes relatively easily to find test plan implementation gaps, when comparing the two models. Often, one would also expect to find design issues with such a comparison. However, for a widely implemented functionality such as UNIX Spawn, that is highly unlikely.

The two resulting models were compared in a similar fashion to the LPM case study, looking for parameters or values missing from the testing model created with the cluster-based test plan functional analysis technology. There was a very high correlation of parameters and values between the two models.

When comparing the two models, both parameter names and value names are rarely identical. However, we found manually matching parameters and values in this case to be easy. The manually created design model contained eleven parameters. Seven of those parameters were found by using our tool on the tests. The remaining four parameters were not mentioned anywhere in the tests themselves. This indicated test gaps.

There were twenty six values in the design model for the parameters that were also part of the test model. The semi-automatic test model detected thirteen values directly. At least three others could easily be deduced. For example, the parameter `program control` had an `On` value. The tester could reasonably assume that it also has an `Off` value. The fact that such values were missing from the analyzed tests was another indication of test gaps.

After a discussion with the SME we learned that the values that our cluster-based test plan functional analysis technology did not encounter appear in tests that were omitted from the technology input. This suggested a potential reorganization of the existing test cases, as the missing tests were assumed to apply mainly to a closely related functionality but not specifically also to the Spawn functionality.

3.3.3 Value. Using the cluster-based test plan functional analysis technology it was possible to relate models created by a human expert from design artifacts with those semi-automatically created from the tests. The technology supported doing this while investing significantly less effort compared with manual modeling. We invested 16 hours in developing the test model using our clustering technology. Based on our past experience in full manual modeling from existing tests and the fact that the test sample contained 472 test cases which without our technology would have to be all manually reviewed, we estimate the effort we invested as around 20% of a similar fully manual effort. This constitutes a significant reduction in the cost of applying combinatorial modeling from test artifacts, hence further increases the overall value of our proposed use cases for CT.

In addition, it was possible to map design model artifacts to test model artifacts, supporting the usage of CT as a means to model and compare the different phase models for the purpose of finding gaps.

4 DISCUSSION

In the following we summarize the main lessons learned and the main points of value identified from our practical experience with CT in the novel use cases presented in this work.

- **Value of CT at review time.** Some interactions resulting from implementing code per the design are very difficult for a reviewer to identify. Extensive software state dependencies can easily be overlooked by even a subject matter expert. By modeling these state based interactions and generating reviews for design with CT, all reviewers benefit regardless of experience level.
- **Value of comparing artifacts.** Comparison of different artifacts that describe the same concepts from different angles and in different ways is a well-known review technique. Unfortunately, it is not often enough used. By applying it in real-world settings we re-confirmed its strength, as it quickly and clearly raised issues resulting from gaps and inconsistencies between the compared artifacts.
- **Providing confidence and reassurance.** Comparison of artifacts is not only a means to find issues but also a means to increase confidence in the quality of the system under test. When consistency is found between parts of compared artifacts, the gained value is twofold. First, it confirms the human understanding of the semantics of the consistent parts. Second, it increases the confidence in the technologies that were used to produce the artifacts, when such were used. The fact that the Spawn test model developed with the assistance of our clustering-based technique mapped well to a different model created by a SME demonstrates the usefulness and effectiveness of the used technology. It also increases the confidence in the quality of the tests that were used as input for the technology.
- **Designer versus tester roles.** CT provides a unique lens to view design requirements. When identifying design interactions that matter to test, the background of the reviewer

matters. Designers do not consider the same factors as testers. We noted completely different lists of meaningful interactions for LPM depending on the role. CT facilitates taking greater advantage of the designer and tester viewpoints.

- **Use of other modeling and review techniques.** Many modeling and review techniques exist, and can be utilized to detect design flaws and test coverage gaps. We found CT to be a generic, effective and natural way to model various aspects of the software based on different types of artifacts, such as requirements, design documents and existing tests. As we point out in Section 2, we used the same methodology and technology to model each aspect, using a different mindset. Other modeling techniques would have to define how to create models based on each type of artifact.
- **Early detection of issues.** There is universal agreement that finding defects as early in the development process as possible is beneficial. Making use of techniques and tools that increase the potential for such early discovery makes good sense. We encourage practitioners to apply combinatorial modeling in concert with existing design and code review methods.

5 RELATED WORK

We are unaware of any work describing the use of combinatorial modeling at design phase for the purpose of design and code review. We are also unaware of work identifying gaps between design and test by comparing combinatorial models derived separately from design and test artifacts.

There is a large body of work on various aspects of combinatorial testing, but we are aware of only a couple of works that discuss tool-assisted CT model extraction from existing artifacts. In [23], we suggest a clustering-based technique that assists in functional analysis of free-text test plans, and report its application to real-world test plans of industrial companies. The results of the analysis are used for different purposes such as test plan automation and management, identification of redundant test cases, and CT model extraction for test generation. In [23], CT models extracted by our technique were validated by a SME review. In this work, we take the technique a step further by extending its use case to identify gaps between design and test, and by validating the extracted model via a comparison to a model created by a SME.

With the exception of [23], we are unaware of work describing tool-assisted CT model extraction from existing test plans. In [18], a rule based approach for deriving CT models from UML activity diagrams is suggested. The diagram is automatically parsed, and different rules are applied to detect parameters, values and constraints. While the use of a fully structured artifact has the potential of extracting more accurate models, the approach also relies on the existence of such structured artifacts, an assumption which in our settings usually does not hold.

[16] suggests a technique for semi-automated CT constraints identification from requirement documents. As opposed to our approach which detects potential model parameters and values, [16] assumes they are already given, and tries to detect combinations of parameters that participate in the same constraint. This is achieved by measuring the distance between values of the parameters in the text of the document – the smaller the distance is, the stronger the relation is assumed to be. The technique does not guarantee that a small distance indeed indicates a joint constraint, and it does not characterize the nature of such a constraint in any way. The approach is evaluated on one case study which was developed in academic settings.

As these technologies mature, they could complement our clustering-based technique by providing assistance to the SME in creating combinatorial models from industrial design artifacts.

6 CONCLUSIONS

In this work we present two novel use cases of combinatorial modeling throughout the development life cycle: generating scenarios for review of design and code, and finding gaps between design and test, with and without tool-assisted model definition. Both use cases were applied to large industrial products for server operation and maintenance. Applying CT at design phase resulted in identifying two design flaws. According to the design team, detection of these problems was challenging even for the best subject matter expert. Fixing those flaws prior to code completion and any testing reflects the benefit from combinatorial modeling at design time. The application of CT for comparing design with test in the first case study revealed test gaps correlated with 20% of field-reported code defects, as well as some design gaps. Comparing design with test models results in pro actively finding potential high impact issues thus avoiding their extremely high cost. CT modeling at an early design stage could have avoided these test gaps and their resulting field problems. In the second case study for comparing design with test, where the design was already well-established, the application of CT via a novel cluster-based technology for functional analysis of test plans resulted in a test model that mapped well to the SME design model, increasing the confidence both in the technology that was used to generate the test model and in the quality of the existing test plan.

In the future, we would like to further assess the overall impact of the design phase CT modeling on CFCC, by collecting quantitative and qualitative experience data. We would also like to include additional case studies for design phase modeling, in which the design is modeled before coding has even started, to maximize the benefit of early detection of design issues. Our future case studies should also include a variety of product types from different domains. We invite other companies to evaluate the impact of proactive and pervasive combinatorial testing on the quality of their products.

We believe our approach to fit nicely with any software development process, including Agile processes. It is left for future work to demonstrate this on a real-world software system.

Another direction we are exploring is to come up with a Return On Investment (ROI) model that quantitatively demonstrates that the application of CT returns more than it costs, and that CT will provide as good a return as other design and code review techniques, when applied at design phase.

Finally, we plan to investigate the applicability of our cluster-based functional analysis technique to other types of artifacts, such as requirement documents and user stories. Extending the technology to support analysis of requirements and design artifacts

would further extend its ability to support the novel CT use cases presented in this work.

ACKNOWLEDGMENTS

The authors thank Aviad Zlotnick, Onn Shehory, and Eileen Tedesco, for exercising their mastery of combinatorial modeling for the CFCC, LPM, and Spawn studies, respectively. We thank Eitan Farchi for spinning the thread for application of combinatorial testing at design phase from which the fabric of the included studies was woven.

REFERENCES

[1] K. Z. Bell. 2006. *Optimizing Effectiveness and Efficiency of Software Testing: A Hybrid Approach.* Ph.D. Dissertation. North Carolina State University. Advisor(s) Vouk, M. A.

[2] K. Z. Bell and M. A. Vouk. 2005. On effectiveness of pairwise methodology for testing network-centric software. In *2005 International Conference on Information and Communication Technology*. 221–235.

[3] K. Burroughs, A. Jain, and R.L. Erickson. 1994. Improved quality of protocol testing through techniques of experimental design. In *SUPERCOMM/ICC*.

[4] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn. 2003. Constructing test suites for interaction testing. In *25th International Conference on Software Engineering, 2003. Proceedings.* 38–48.

[5] M. B. Cohen, J. Snyder, and G. Rothermel. 2006. Testing across configurations: implications for combinatorial testing. *SIGSOFT Softw. Eng. Notes* (2006).

[6] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. 1999. Model-Based Testing in Practice. In *Proc. 21st Intl. Conf. on Software Engineering (ICSE'99)*. 285–294.

[7] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDIR*. 226–231.

[8] M. Grindal, B. Lindström, J. Offutt, and S. F. Andler. 2006. An evaluation of combination strategies for test case selection. *Empirical Softw. Eng.* (2006).

[9] A. Hayardeny, S. Fienblit, and E. Farchi. 2004. Concurrent and Distributed Desk Checking. In *Parallel and Distributed Systems: Testing and Debugging (PADTAD - 2). In conjunction with 18th International Parallel and Distributed Processing Symposium (IPDPS)*.

[10] IBM FOCUS [n. d.]. IBM Functional Coverage Unified Solution (IBM FOCUS). http://researcher.watson.ibm.com/researcher/view_group.php?id=1871. ([n. d.]).

[11] C. Jones. 2008. *Applied Software Measurement: Global Analysis of Productivity and Quality* (3 ed.). McGraw-Hill Osborne Media.

[12] D. R. Kuhn, Raghu N. Kacker, and Y. Lei. 2013. *Introduction to Combinatorial Testing.* Chapman & Hall/CRC.

[13] D. R. Kuhn and M. J. Reilly. 2002. An Investigation of the Applicability of Design of Experiments to Software Testing. 27th NASA/IEEE Software Engineering Workshop, NASA Goddard Space Flight Center. (2002).

[14] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, Jr. 2004. Software Fault Interactions and Implications for Software Testing. *IEEE Trans. Softw. Eng.* 30, 6 (2004), 418–421.

[15] G. Myers. 1979. *The Art of Software Testing.* ohn Wiley and Sons.

[16] H. Nakagawa and T. Tsuchiya. 2015. Towards Automatic Constraints Elicitation in Pair-Wise Testing Based on a Linguistic Approach: Elicitation Support Using Coupling Strength. In *Requirements Engineering and Testing (RET)*. 34–36.

[17] C. Nie and H. Leung. 2011. A Survey of Combinatorial Testing. *ACM Comput. Surv.* 43, 2 (2011), 11:1–11:29.

[18] P. Satish, K. Sheeba, and K. Rangarajan. 2013. Deriving Combinatorial Test Design Model from UML Activity Diagram. In *Software Testing, Verification and Validation Workshops (ICSTW)*. 331–337.

[19] I. Segall, R. Tzoref-Brill, and E. Farchi. 2011. Using Binary Decision Diagrams for Combinatorial Test Design. In *ISSTA*.

[20] D. R. Wallace and D. R. Kuhn. 2001. Failure Modes in Medical Device Software: an Analysis of 15 Years of Recall Data. In *ACS/ IEEE International Conference on Computer Systems and Applications*. 301–311.

[21] A. W. Williams. 2000. Determination of Test Configurations for Pair-Wise Interaction Coverage. In *TestCom*.

[22] P. Wojciak and R. Tzoref-Brill. 2014. System Level Combinatorial Testing in Practice – The Concurrent Maintenance Case Study. In *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. 103–112.

[23] M. Zalmanovici, O. Raz, and R. Tzoref-Brill. 2016. Cluster-Based Test Suite Functional Analysis. *24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)* (2016).

[24] Z. Zhang and J. Zhang. 2011. Characterizing Failure-causing Parameter Interactions by Adaptive Testing. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*. 331–341.