Modify, Enhance, Select: Co-Evolution of Combinatorial Models and Test Plans

Rachel Tzoref-Brill
School of Computer Science, Tel Aviv University
and IBM Research, Israel

Shahar Maoz
School of Computer Science, Tel Aviv University, Israel

ABSTRACT

The evolution of software introduces many challenges to its testing. Considerable test maintenance efforts are dedicated to the adaptation of the tests to the changing software. As a result, over time, the test repository may inflate and drift away from an optimal test plan for the software version at hand. Combinatorial Testing (CT) is a well-known test design technique to achieve a small and effective test plan. It requires a manual definition of the test space in the form of a combinatorial model, and then automatically generates a test plan design, which maximizes the added value of each of the tests. CT is considered a best practice, however its applicability to evolving software is hardly explored.

In this work, we introduce a first co-evolution approach for combinatorial models and test plans. By combining three building blocks, to minimally modify existing tests, to enhance them, and to select from them, we provide five alternatives for co-evolving the test plan with the combinatorial model, considering tradeoffs between maximizing fine-grained reuse and minimizing total test plan size, all while meeting the required combinatorial coverage.

We use our solution to co-evolve test plans of 48 real-world industrial models with 68 version commits. The results demonstrate the need for co-evolution as well as the efficiency and effectiveness of our approach and its implementation. We further report on an industrial project that found our co-evolution solution necessary to enable adoption of CT with an agile development process.

CCS CONCEPTS

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

KEYWORDS

Combinatorial Testing, Software Evolution

ACM Reference Format:
Rachel Tzoref-Brill and Shahar Maoz. 2018. Modify, Enhance, Select: Co-Evolution of Combinatorial Models and Test Plans. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3236024.3236067

1 INTRODUCTION

Software systems continuously change during development and maintenance to support evolving requirements and design. Software evolution is intensified by modern paradigms such as agile development [29] and continuous delivery [17], which advocate incremental software changes within shorter development cycles and frequent deliveries. Evolving software imposes challenges for its testing procedures. Tests designed for a certain version of the System Under Test (SUT) might be invalidated following software changes and have to be either repaired or discarded. New tests need to be created for newly added functionality. When test generation is manual, discarding tests as well as creating new ones is expensive. Test repair involves manual analysis and might be labor intensive as well. Over time, incremental test accumulation creates a technical debt, both because it involves continuous manual test maintenance effort, and because the test repository inflates and drifts further away from an optimal set of tests for the SUT version at hand.

Combinatorial Testing (CT) is a well-known testing technique to achieve a small and effective test plan [5, 9, 11, 15, 21, 39, 40]. The rationale behind CT and the key for its effectiveness is the empirical observation that in most cases, the occurrence of a bug depends on the interaction between a small number of features of the SUT [1, 2, 23, 24, 38, 41]. For example, according to [24], all possible pairs of parameter values can typically detect 47% to 97% of the bugs in an SUT. CT translates this observation to a coverage criterion as follows. It requires a manual definition of the test space in the form of a *combinatorial model*, consisting of a set of parameters, their respective values, and constraints on the value combinations. 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 set of valid tests, termed a *test plan*, which covers all valid value combinations of every t parameters, where t is usually a user input. Such a test plan is said to achieve 100% *t-way interaction coverage*. A significant *combinatorial reduction* is achieved in the size of the resulting test plan (compared to manually designed test plans for example) because the tests generated by the CT algorithm are very different from each other, maximizing their added value – each of them covers as many unique t-way value tuples as possible. Note that tests produced by the algorithm are parameter-value assignments. Generating executable tests from them is often a manual effort.

While there is much evidence for the effectiveness of CT on a single version of an SUT with a single combinatorial model, its applicability to evolving software is hardly explored. Moreover, while many academic and industrial tools for CT exist, e.g., PICT [10], ACTS [25], CASA [13], and AETG [7], none supports model and test plan evolution. Indeed, applying CT to an evolving SUT is non-trivial. Straightforward naive approaches will typically not work best due to the tradeoff between the total test plan size and the number of new tests that need to be developed. Specifically, on the one hand, repeated application of CT that accumulates the resulting tests from one version to the next will unnecessarily inflate the total test plan size. On the other hand, discarding all tests from previous versions and starting each time from scratch is wasteful, since it avoids test reuse and thus unnecessarily increases test generation effort. A middle ground approach is called for.

In this work, we present a first approach for co-evolution of combinatorial models and test plans. Our approach is based on the observation that in CT, software evolution translates into model definition updates, and these updates together with the structured tabular form of the combinatorial test plan consist a unique opportunity to assist in efficient test plan evolution. Given a combinatorial model and an existing combinatorial test plan from a previous version of the model as input, our solution to co-evolution consists of three main building blocks: (1) **Modifying** existing tests to match the new model version while maximizing the gain in t-way interaction coverage; (2) **Enhancing** the existing test plan with a small set of new tests to reach 100% t-way interaction coverage of the new model version; and (3) **Selecting** a subset of the resulting test plan that retains 100% t-way interaction coverage, by eliminating redundant tests that no longer contribute to the coverage.

To achieve efficient modification of tests, we apply an algorithm that automatically locates minimal value combinations in the existing tests that are invalidated by the changes made in the new model version. After identification, the algorithm replaces these values with new values that are consistent with the new model version and maximize the overall t-way coverage of the test plan. Finally, the modification algorithm appends new values to existing tests for parameters that were added in the new model version to represent newly added SUT functionality.

Test plan enhancement computes the t-way coverage of the existing tests, and creates new tests that augment them to reach overall 100% t-way coverage. To achieve this, it uses a variant of a CT algorithm that receives a seed of tests as its input. Seeding was suggested in [7] as a way for the engineer to specify tests that must appear in the solution. We provide the existing tests from the previous model version as the seed to the CT algorithm.

Finally, for test plan selection we use interaction-based test suite minimization (ITSM) [3]. Given a test plan, ITSM selects a subset of the tests that preserves the same t-way interaction coverage as the original test plan. ITSM was suggested in [3] in the context of a single model version, where it was used to eliminate redundancy from manually created test plans. We observe that ITSM is also very useful in the context of evolution to eliminate redundancy originating from model modifications (especially functionality removal). Thus, after modification and enhancement we apply the ITSM algorithm to the resulting test plan, to select a subset that preserves 100% t-way coverage.

Considering that different SUT settings have different pain points, the efforts associated with test generation and total test plan size vary, and depend on various factors such as SUT complexity, test execution time, and the extent of automation embedded into the test procedures. To accommodate for this variability, we use the above three building blocks, modify, enhance, and select, to present the engineer with five different alternatives for co-evolving the test plan with the model. We note that each of these alternatives produces a test plan with 100% t-way coverage of the current model version. The alternatives are as follows.

(1) **CT from Scratch (CT)** results in potentially minimal test plan size, but in a maximal number of new tests as there is no reuse of existing ones.
(2) **Enhancement (EN)** maximizes the number of reused tests but results in the largest test plan size. Tests that are invalidated by model changes are discarded, and new tests are created to replace the lost functionality, as well as to cover newly added functionality.
(3) **Enhance-Select (ES)** is similar to Enhancement, but also applies selection to the resulting test plan, hence its total size is potentially smaller – it typically contains the same number of new tests as Enhancement, but with fewer existing ones.
(4) **Modify-Enhance-Select (MES)** applies all three building blocks of the solution, and results in potentially the least number of new tests to create. Invalidated tests are repaired in a way that maximizes the gain in t-way interaction coverage.
(5) **Modify-Enhance-Select without Repair (MESW)** is a variant of Modify-Enhance-Select which allows to discard invalidated tests instead of repairing them, hence the modification algorithm only appends new values to existing tests for added parameters.

We implemented our co-evolution solution within the industrial-strength commercial CT tool IBM Functional Coverage Unified Solution (IBM FOCUS) [18, 35]. To scale the computation to real-world model sizes, we use an efficient representation of the sets of valid tests, based on Binary Decision Diagrams [4], a compact data structure for representing and manipulating Boolean functions. Thanks to our efficient implementation, FOCUS can compute the different co-evolution alternatives almost instantaneously. Therefore, the engineer does not have to a-priori decide which alternative to use. Rather, she can examine the resulting alternative test plans and choose the one that best suits the current needs.

Note that there are rules of thumb suggesting which alternative to use in which settings, e.g., if executing each single test takes much time then total test plan size is most important to minimize; if generating executable tests out of CT abstract tests is manual labor-intensive task then maximizing reuse (while paying in total size) may be the target. However, we deliberately do not present explicit rules because our tool can easily show all alternatives; engineers should consider the tradeoffs per their setting.

The question of oracle definition in CT (without addressing model evolution) is surveyed in [20]. Approaches for test oracles in CT are categorized as non-automated, implicit, derived, and specified. The latter allows the engineer to specify rules in a formal specification language to determine the expected results for each test case; this is the approach used by most CT tools, as well as by ours. IBM FOCUS supports defining test snippets and their expected results and associating them with specific parameter value combinations. The complete oracle for each test is combined from the relevant test snippets, according to its value combinations. This allows also for reuse of snippets between different tests. In our present work, the cost of defining and updating oracles is included in the cost of defining new and modified tests, and thus in the corresponding tradeoff with total number of tests to maintain and execute.

To evaluate the applicability of our work to real-world model evolution, we applied our co-evolution technique to 48 real-world industrial models with a total of 116 versions and 68 commits (2-5 versions per model). The models and their versions were obtained from IBM. Our analysis shows that the modify-enhance-select alternative achieves a significant reduction in the number of new tests that are required, yet pays very little in total test plan size. Hence, it is an efficient middle ground approach between the two extreme alternatives. Performance wise, our approach provides excellent running times in most cases, and acceptable ones in all cases.

We further report on experience from a real-world industrial project where our co-evolution technique was used by engineers working in agile development mode. Since in this project, the tests automatically generated by IBM FOCUS are manually translated into test automation scripts, test reuse is of high priority. The feedback from the engineers following their experience with our co-evolution technique is that the modify-enhance-select alternative is the most valuable one and in fact was necessary to enable their adoption of CT in conjunction with their agile process.

2 BACKGROUND

We provide background on combinatorial models and test plans and their semantics and on the use of binary decision diagrams to represent models and generate test plans.

Combinatorial Models and Test Plans. A combinatorial model is defined as follows. Let $P = \{p_1, \ldots, p_n\}$ be a labelled set of parameters, $V = \{V_1, \ldots, V_n\}$ a labelled set of finite value sets, where V_i is the set of values for p_i, and C a set of Boolean propositional constraints over P. A test $(v_1, \ldots, v_m)$, where $\forall_i,\ v_i \in V_i$, is a tuple of assignments to the parameters in P. The semantics used in practice by CT tools [32] is Boolean semantics. In this semantics, a valid test is a test that satisfies all constraints in C. The semantics of the model is the set of all its valid tests, denoted by $S(P, V, C)$.

A combinatorial test plan T is a set of valid tests so that $T \subseteq S(P, V, C)$. The interaction level t is an integer so that $0 < t \leq |P|$. The t-way interaction coverage of T is defined as the percentage of parameter-value tuples of length t that appear in at least one test in T out of all valid tuples of length t in $S(P, V, C)$. The definition of t-way interaction coverage can be naturally extended to *variable strength generation* [7, 8], which allows requesting different interaction levels for different subsets of the parameters. Alternative interaction coverage definitions appear in [22].

Given a model $S(P, V, C)$ and an interaction level t, a CT algorithm automatically produces a small as possible test plan that achieves 100% t-way interaction coverage. The construction of such a minimal test plan is an NP-complete problem [26, 36]. Numerous CT algorithms exist, and none is superior in general to all others.

Using Binary Decision Diagrams to Represent Combinatorial Models and Generate Test Plans. In [35], Segall et al. presented a compact representation of combinatorial models using Binary Decision Diagrams (BDDs). BDDs [4] are a compact data structure for representing and manipulating Boolean functions, commonly used in formal verification [6] and in logic synthesis [27]. [35] utilizes the efficient computation of Boolean operations on BDDs such as negation, conjunction, and disjunction, to compute the BDD representing the set of valid tests from the user-specified constraints. The set of invalid tests in the model is represented using the conjunction of the BDDs for each of the constraints. Multi-valued parameters are handled using standard Boolean encoding and reduction techniques to BDDs [30]. The set of valid tests is represented by the negation of the BDD for the invalid tests, conjunct with a BDD that represents the legal multi-valued to Boolean encodings of the parameter values. This BDD-based representation of the combinatorial model is the basis for the implementation of our co-evolution solution.

[35] further presents a BDD-based algorithm to generate a test plan with 100% t-way coverage. Since the implementation of our co-evolution solution relies on a variant of this algorithm, we present in Algorithm 1 relevant excerpts of the original algorithm from [35].

```
input  : Coverage requirements, given as a set $C_R$ of
         tuples of parameters to cover.
         The BDD $Valid$ of all valid tests.
1  Init: for $c \in C_R$ do $uncov(c) \leftarrow Proj_c(Valid)$;
2  while $C_R \neq \emptyset$ do
3      $Collected \leftarrow Valid$;
4      Sort $C_R$ in decreasing order of $sat(uncov(c))$;
5      for $c \in C_R$ do
6          if $(Collected \wedge uncov(c)) \neq FALSE$ then
7              $Collected \leftarrow Collected \wedge uncov(c)$;
8          end
9      end
10     $chosen \leftarrow$randSat($Collected$) appendResult($chosen$);
11     for $c \in C_R$ do
12         $uncov(c) \leftarrow uncov(c) \wedge \neg chosen$;
13         if $uncov(c) = FALSE$ then
14             $C_R \leftarrow C_R \setminus c$;
15         end
16     end
17 end
```

Algorithm 1: BDD-Based CT, excerpt of the original algorithm from [35]

In each iteration, the algorithm selects a single test to add to the test plan as follows. It conjuncts the BDD $Valid$ representing the set of valid tests with the BDDs in C_R representing the sets of uncovered parameter-value tuples in descending order of the BDD sizes (line 7). It then randomly selects a satisfying assignment *chosen* out of the resulting BDD, and adds it to the test plan (line 10). *chosen* represents a test that covers as many new tuples as possible. The tuples covered by *chosen* are removed from the BDDs of the uncovered tuples (line 12). The algorithm proceeds until no uncovered tuples are left.

In Section 4, we will refer to the above algorithm and describe the changes we made to it in order to adapt it to support co-evolution.

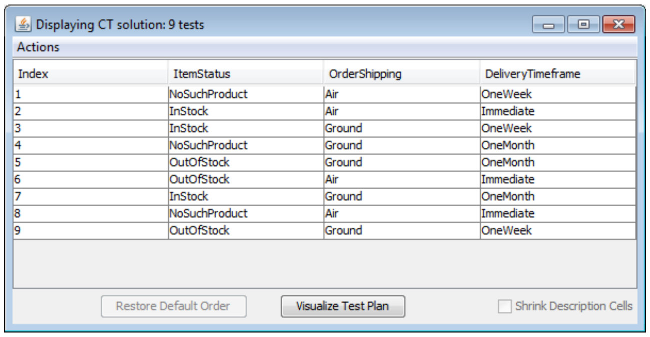
3 RUNNING EXAMPLE AND OVERVIEW

We start off with an example and overview of our work. The presentation here is semi-formal. Formal definitions appear in Section 4.

Table 1: Example on-line shopping model

Parameter	Values
ItemStatus (IS)	InStock, OutOfStock, NoSuchProduct
OrderShipping (OS)	Air, Ground
DeliveryTimeframe (DT)	Immediate, OneWeek, OneMonth

Constraints
`DT = Immediate` $\longrightarrow$ `OS = Air`
`DT = OneMonth` $\longrightarrow$ `OS = Ground`


Displaying CT solution: 9 tests

Actions

Index	ItemStatus	OrderShipping	DeliveryTimeframe
1	NoSuchProduct	Air	OneWeek
2	InStock	Air	Immediate
3	InStock	Ground	OneWeek
4	NoSuchProduct	Ground	OneMonth
5	OutOfStock	Ground	OneMonth
6	OutOfStock	Air	Immediate
7	InStock	Ground	OneMonth
8	NoSuchProduct	Air	Immediate
9	OutOfStock	Ground	OneWeek

Figure 1: A pairwise test plan for on-line shopping model version V1.

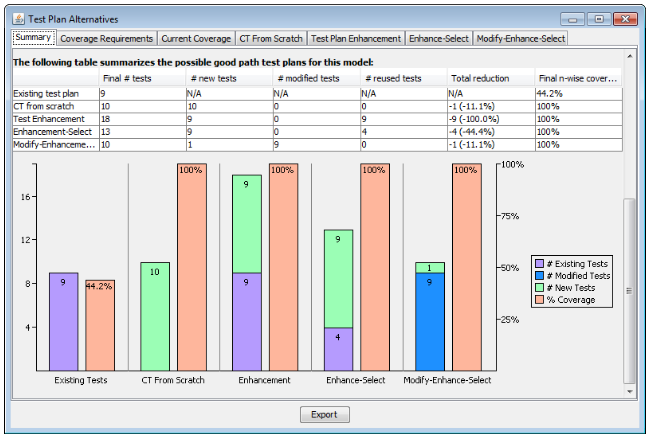
Table 1 shows the parameters, values, and constraints of a combinatorial model for an on-line shopping system, which we use as a running example, borrowed from [37]. The model defines the test space and which tests in it are valid. For example, the test (IS = `InStock`, OS = `Air`, DT = `Immediate`) is valid, while the test (IS = `InStock`, OS = `Ground`, DT = `Immediate`) is invalid.

A pairwise test plan (one of many possible) for this model appears in Figure 1. The test plan is automatically generated by IBM FOCUS. It contains 9 tests, and achieves pairwise coverage of the model because every pair of values for every pair of parameters that is not excluded by the constraints appears in at least one of the 9 tests. Let us assume that a engineer manually translated these 9 tests into 9 executable tests.

Below we follow a series of updates to the model, inspired by similar updates we have seen in the evolution of real-world models. These updates account for changes in the SUT following evolving requirements or design. We describe the updates and demonstrate how our co-evolution solution handles them to co-evolve the test plan. The updates are relatively small and local. We use them to demonstrate the basic principles of our technique.

From V1 to V2. Following the addition of a built-in payment function to the system, a engineer added a new parameter named `PaymentMethod (PM)` with values `CreditCard`, `PayPal`, and `GiftVoucher`, and committed a second model version, model version V2. Once the model was updated, the pairwise test plan needs to co-evolve with it in order to be up-to-date with the test space changes. Figure 2 depicts the result of our co-evolution technique as produced by IBM FOCUS on the pairwise test plan from Figure 1 and the on-line shopping model version V2.

On the left-most column, we see that the pairwise test plan from model version V1 achieves only 44.2% pairwise coverage in model version V2. This is because all the interactions of the newly added payment function with the other parameters are not covered by the


Test Plan Alternatives

The following table summarizes the possible good path test plans for this model:

	Final # tests	# new tests	# modified tests	# reused tests	Total reduction	Final n-wise cover...
Existing test plan	9	N/A	N/A	N/A	N/A	44.2%
CT from scratch	10	10	0	0	-1 (-11.1%)	100%
Test Enhancement	18	9	0	9	-9 (-100.0%)	100%
Enhancement-Select	13	9	0	4	-4 (-44.4%)	100%
Modify-Enhanceme...	10	1	9	0	-1 (-11.1%)	100%

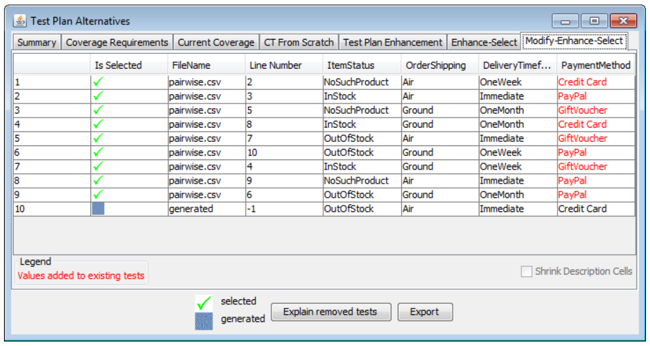
Figure 2: Co-evolution alternatives for pairwise test plan version V1 with on-line shopping model version V2.

existing test plan. To evolve the test plan and reach 100% pairwise coverage in model version V2 while optimizing the gain in pairwise coverage, we present 4 alternatives, in order from left to right:

- **CT from Scratch.** If we discard the entire test plan from V1, we need to create 10 new tests. They are detailed in the 'CT from Scratch' tab.
- **Enhancement.** If we keep the entire test plan from V1 as is, we need to add 9 new tests to reach 100% pairwise coverage. The total size of the test plan is 18 (9 new, 9 existing). The 9 new tests are detailed in the 'Test Plan Enhancement' tab.
- **Enhance-Select.** After enhancement by 9 new tests, we can remove 5 tests from the original V1 test plan and still reach 100% pairwise coverage. This leads to a test plan of size 13 (9 new, 4 existing). The 13 tests are detailed in the 'Enhance-Select' tab.
- **Modify-Enhance-Select.** If we modify the 9 existing tests and add to them the payment function, we need to create only one new test. The total size of the test plan is 10 (9 modified, 1 new). The 10 tests are detailed in the 'Modify-Enhance-Select' tab. The values that were added to the existing tests are marked with red, as depicted in Figure 3.

When examining the different alternatives, it seems that in this case the 'Modify-Enhance-Select' alternative is the most cost-effective, since it results in the smallest number of tests in total, similarly to 'CT from Scratch', however with only one new test, compared to 10 new tests in 'CT from Scratch'. Let us assume that the engineer decided to choose this alternative and transform it into an executable test plan, by modifying the 9 existing tests and creating one new test as instructed by IBM FOCUS.

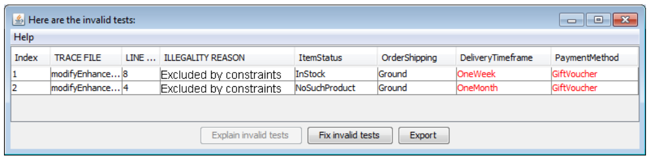
From V2 to V3. Following a change in the system's policy, gift vouchers are accepted as a payment method only when the delivery time frame is immediate. To reflect this policy change, the engineer added a new constraint to the model, PM = `GiftVoucher` $\longrightarrow$ DT = `Immediate`, and committed a third model version V3. When trying to co-evolve the test plan from V2, IBM FOCUS informs that some tests are invalidated in the new model version V3. Three options are provided: (1) View invalidated tests, (2) Discard invalidated tests



	Is Selected	FileName	Line Number	ItemStatus	OrderShipping	DeliveryTimef...	PaymentMethod
1	✓	pairwise.csv	2	NoSuchProduct	Air	OneWeek	Credit Card
2	✓	pairwise.csv	3	InStock	Air	Immediate	PayPal
3	✓	pairwise.csv	5	NoSuchProduct	Ground	OneMonth	GiftVoucher
4	✓	pairwise.csv	8	InStock	Ground	OneMonth	Credit Card
5	✓	pairwise.csv	7	OutOfStock	Air	Immediate	GiftVoucher
6	✓	pairwise.csv	10	OutOfStock	Ground	OneWeek	PayPal
7	✓	pairwise.csv	4	InStock	Ground	OneWeek	GiftVoucher
8	✓	pairwise.csv	9	NoSuchProduct	Air	Immediate	PayPal
9	✓	pairwise.csv	6	OutOfStock	Ground	OneMonth	PayPal
10	▨	generated	-1	OutOfStock	Air	Immediate	Credit Card

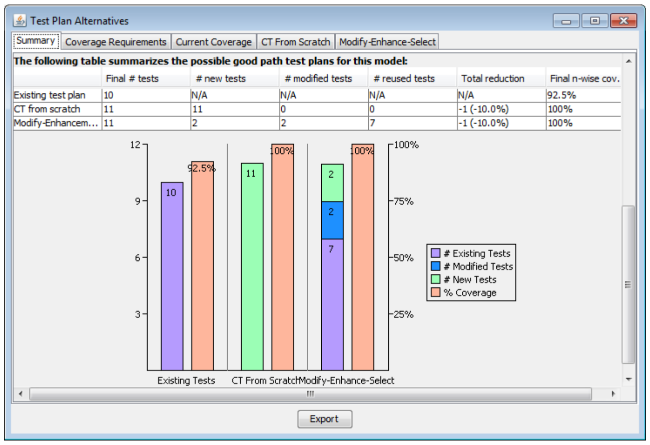
Legend
Values added to existing tests

Figure 3: The Modify-Enhance-Select alternative for pairwise test plan version V1 and on-line shopping model version V2. Added values are marked with red. The 'Is Selected' column indicates whether the test is a new one or an existing one.



Index	TRACE FILE	LINE ...	ILLEGALITY REASON	ItemStatus	OrderShipping	DeliveryTimeframe	PaymentMethod
1	modifyEnhance...	8	Excluded by constraints	InStock	Ground	OneWeek	GiftVoucher
2	modifyEnhance...	4	Excluded by constraints	NoSuchProduct	Ground	OneMonth	GiftVoucher

Figure 4: Tests from the test plan produced in version V2 that were invalidated by changes made to version V3. Minimal invalidated combinations are automatically detected and marked with red.



The following table summarizes the possible good path test plans for this model:

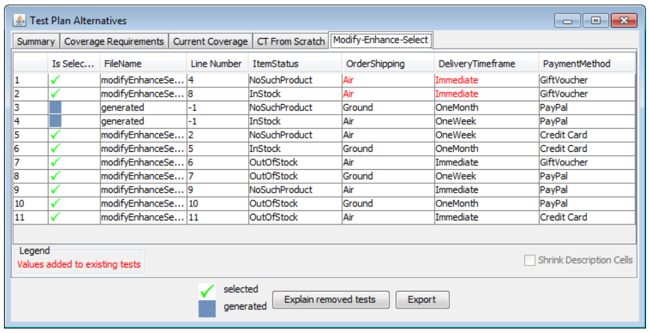
	Final # tests	# new tests	# modified tests	# reused tests	Total reduction	Final n-wise cov.
Existing test plan	10	N/A	N/A	N/A	N/A	92.5%
CT from scratch	11	11	0	0	-1 (-10.0%)	100%
Modify-Enhancem...	11	2	2	7	-1 (-10.0%)	100%

Figure 5: Co-evolution alternatives for pairwise test plan version V2 with on-line shopping model version V3, when existing invalidated tests are repaired.

and continue without them, and (3) Fix invalidated tests to match current model version.

If the engineer chooses to view the invalidated tests (opt. #1), she will see the results as in Figure 4. Note that IBM FOCUS automatically detects the reason for invalidity of each test, and presents it together with minimal invalidated combinations (marked with red). Such combinations may be easy to manually identify in a simple toy model, however it quickly becomes labor intensive and error prone to manually detect them in a more complex model with many constraints, hence automated detection is of much value.

If the engineer decides to discard the invalid tests and perform the co-evolution without them (opt. #2), then the 2 invalid tests are



	Is Selec...	FileName	Line Number	ItemStatus	OrderShipping	DeliveryTimeframe	PaymentMethod
1	✓	modifyEnhanceSe...	4	NoSuchProduct	Air	Immediate	GiftVoucher
2	✓	modifyEnhanceSe...	8	InStock	Air	Immediate	GiftVoucher
3	▨	generated	-1	NoSuchProduct	Ground	OneMonth	PayPal
4	▨	generated	-1	InStock	Air	OneWeek	PayPal
5	✓	modifyEnhanceSe...	2	NoSuchProduct	Air	OneWeek	Credit Card
6	✓	modifyEnhanceSe...	5	InStock	Ground	OneMonth	Credit Card
7	✓	modifyEnhanceSe...	6	OutOfStock	Air	Immediate	GiftVoucher
8	✓	modifyEnhanceSe...	7	OutOfStock	Ground	OneWeek	PayPal
9	✓	modifyEnhanceSe...	9	NoSuchProduct	Air	Immediate	PayPal
10	✓	modifyEnhanceSe...	10	OutOfStock	Ground	OneMonth	PayPal
11	✓	modifyEnhanceSe...	11	OutOfStock	Air	Immediate	Credit Card

Legend
Values added to existing tests

Figure 6: The Modify-Enhance-Select alternative for pairwise test plan version V2 and on-line shopping model version V3. Repaired values are marked with red.

discarded, leaving the existing test plan with 8 tests that achieve 87.5% pairwise coverage in model version V3. CT from scratch results in 11 new tests, however if the 8 existing tests are kept, only 4 new tests are needed by all other alternatives. The price for keeping the existing tests is low, a total of 12 tests instead of 11.

Finally, if the engineer opts to repair the invalidated tests (opt. #3), then as shown in Figure 5, the existing test plan now contains all 10 tests and achieves 92.5% pairwise coverage. The 'Modify-Enhance-Select' alternative now results in only 11 tests, and only 2 of them are new. The other 2 tests are repaired according to the new value assignments that appear in the 'Modify-Enhance-Select' tab, marked with red, as shown in Figure 6. 'CT from Scratch' still results in the same 11 tests as before, since it does not consider the existing test plan. As the other alternatives do not involve modifications to the existing test plan, they are not shown in this case.

4 CO-EVOLUTION OF COMBINATORIAL MODELS AND TEST PLANS

We now formally present our co-evolution solution for combinatorial models, followed by a description of our algorithm for computing it. We use the notations defined in Section 2 and demonstrate the ideas on the running example presented in Section 3.

4.1 Co-Evolution Definitions

Definition 4.1. **A combinatorial co-evolution problem** receives (1) an old combinatorial test plan $T \subseteq S(P,V,C)$, (2) a new combinatorial model $S(P',V',C')$, and (3) a set of interaction coverage requirements C_R which includes tuples of parameters to cover[1]. A solution is a new test plan $T' \subseteq S(P',V',C')$ so that T' achieves 100% interaction coverage of C_R in $S(P',V',C')$.

We address the co-evolution problem defined above by providing several alternative solutions, each using a different series of basic transformations on T.

We now define the set of basic transformation functions that we compose later to define our 5 co-evolution alternatives. To this end, T is treated as a two dimensional matrix, where $T(j)$ refers to test number j in T, and $T(j,i)$ refers to the value assignment to parameter p_i in $T(j)$. For a set of indices $K = \{k_1,\ldots,k_m\}$, $T(j,K)$ refers to the value assignment of $T(j)$ to the parameters $\{p_{k_1},\ldots,p_{k_m}\}$.

[1]This is the more generic form of coverage requirements, which supports variable-strength test generation rather than only a single interaction level t.

We also define a special character $\perp$ representing an uninitialized value, i.e., if $T(j,i) = \perp$ then p_i is not assigned with any value in $T(j)$. We term $T(j)$ as a *partial test* if $\exists j : T(j) = \perp$. We term it as a *complete test* if $\forall j : T(j) \neq \perp$. Since the new model $S(P', V', C')$ and coverage requirements C_R are inputs common to all functions, for simplicity we omit them from the formal definitions.

Definition 4.2 (Enhancement function). $F_{EN} : S(P,V,C) \rightarrow S(P,V,C)$. This function creates a small as possible test plan $F_{EN}(T)$ so that $T \subseteq F_{EN}(T)$ and $F_{EN}(T)$ achieves 100% interaction coverage of C_R in $S(P,V,C)$.

Definition 4.3 (Selection function). $F_{SEL} : S(P,V,C) \rightarrow S(P,V,C)$. This function selects a small as possible subset $F_{SEL}(T) \subseteq T$ so that $F_{SEL}(T)$ achieves the same interaction coverage of C_R as T in $S(P,V,C)$.

Definition 4.4 (Projection function). $F_{PROJ} : S(P,V,C) \rightarrow S(P \cap P', V, C)$. This function eliminates from T the set of parameters $P \setminus P'$ by projecting T on $P \cap P'$.

Definition 4.5 (Discard function). $F_{DIS} : S(P,V,C) \rightarrow S(P,V',C')$. This function discards invalid tests from T as follows:
$F_{DIS}(T) \leftarrow T \setminus \{T(i) \mid T(i) \notin S(P,V',C')\}$.

Definition 4.6 (Repair function). $F_{REP} : S(P,V,C) \rightarrow S(P, V \cap V', C')$. This function eliminates from T the sets of values $V_i \setminus V_i{}'$ as follows. If $T(j,i) \in V_i \setminus V_i{}'$ then $T(j,i) \leftarrow \perp$. In addition it eliminates from T value combinations that are invalid in $S(P,V,C')$ as follows. For each $T(j)$, $j = 0, \ldots, |T| - 1$, for each $K \subseteq \{0, \ldots, |P| - 1\}$, if $sat(T(j,K), C') = FALSE$ then $\exists k \in K : T(j,k) \leftarrow \perp$.

Definition 4.7 (Modify function). $F_{MOD} : S(P,V,C) \rightarrow S(P \cup P', V, C)$. This function modifies T as follows. It first extends T by appending the $\perp$ to each $T(j)$ k times, where $k = |P' \setminus P|$. Next, for each $T(i,j)$, if $T(i,j) = \perp$ then $T(i,j) \leftarrow v$, where $v \in V_j$. The selection of v aims at maximizing the overall interaction coverage of $F_{MOD}(T)$ w.r.t. $S(P \cup P', V, C)$ and C_R.

Finally, we define the 5 co-evolution alternatives that we provide. The inputs to our solution are an old test plan $T \subseteq S(P,V,C)$, a new model $S(P',V',C')$, and interaction coverage requirements C_R.

Definition 4.8 (Combinatorial co-evolution solution alternatives).
(1) **CT from Scratch (CT)** is defined as $F_{EN}(\emptyset)$.
(2) **Enhancement (EN)** is defined as $F_{EN}(F_{DIS}(F_{PROJ}(T)))$.
(3) **Enhance-Select (ES)** is defined as
$F_{SEL}(F_{EN}(F_{DIS}(F_{PROJ}(T))))$.
(4) **Modify-Enhance-Select (MES)** is defined as
$F_{SEL}(F_{EN}(F_{MOD}(F_{REP}(F_{PROJ}(T)))))$.
(5) **Modify-Enhance-Select without Repair (MESW)** is defined as $F_{SEL}(F_{EN}(F_{MOD}(F_{DIS}(F_{PROJ}(T)))))$.

For example, in our example in Section 3, when moving from V1 to V2, the MES alternative first applies the projection and repair functions that in this case do not change T because no parameters, values, or value combinations were removed in the second model version. It then applies the modification function which appends a $\perp$ value to the 9 tests in T for the new `PM` parameter, followed by replacement of the $\perp$ values with values out of {`CreditCard`, `PayPal`, `GiftVoucher`} while aiming at increasing the overall pairwise coverage of the 9 tests w.r.t model version V2. Next, the enhancement function adds one new test so that the 10 tests collectively reach 100% pairwise coverage w.r.t model version V2. Finally, the selection function does not remove any tests, since all of them are needed to achieve 100% pairwise coverage.

We note that removing an entire parameter from the model (a case which does not appear in our example) corresponds to functionality that was removed from the SUT. This functionality needs to be removed from the tests as well. Since removal is a simple action compared to repair or addition of functionality, all our co-evolution alternatives (except for the CT alternative) apply it via the F_{PROJ} function to T as a first basic transformation action.

4.2 Computing the Co-Evolving Test Plans

We use a BDD-based implementation for our co-evolution solution. We adapt the BDD-based CT algorithm from [35], described in Section 2 as Algorithm 1. We also use the ITSM algorithm for test plan minimization [3]. In addition, we introduce a new BDD-based algorithm for identifying minimal invalid combinations in a combinatorial test.

In the implementation of the different basic transformation functions, each of the functions is given the following inputs:

(1) The new combinatorial model $S(P',V',C')$;
(2) A BDD $Valid$ representing the set of valid tests in $S(P',V',C')$;
(3) The interaction coverage requirements C_R; and
(4) The old test plan T in the form of a two dimensional matrix, where $T(j,i)$ is the value assigned to existing test number j for parameter p_i.

Each function returns a transformed test plan T' which is given as part of the input list to the next function in the composition.

Projection implementation. To implement F_{PROJ}, we skip the columns i in T for which $p_i \notin P'$.

Discard implementation. To implement F_{DIS}, we iterate over the values of $T(j)$. If we identify a $T(j,i)$ not in $V_i{}'$, we remove $T(j)$ from T. Otherwise, we create a BDD TB_j for $T(j)$ by conjuncting the BDDs of each of its parameter-value assignments. If $TB_j \wedge Valid = FALSE$, we remove $T(j)$ from T.

Enhancement implementation. To implement F_{EN} we use seeding [7], a well-known concept in CT to initialize a solution set with tests that must appear in it. In this case, since we only enhance the seed with new tests and do not modify it, the required adaptation of Algorithm 1 is straightforward. First, we compute an array of BDDs cov so that for each parameter tuple $c \in C_R$, $cov(c)$ represents the set of value tuples of c covered by T. Then, we replace line 1 with the line:

for $c \in C_R$ **do** $uncov(c) \leftarrow Proj_c(Valid) \wedge \neg cov(c)$;

Thus, combinations that are covered by T are excluded from the coverage requirements. The CT algorithm then proceeds as usual to produce a small set of tests S that achieve 100% coverage of the modified coverage requirements. The resulting test plan is $S \cup T$.

Repair implementation. To implement F_{REP}, we first eliminate invalid values by iterating over the values of $T(j)$ and uninitializing every $T(j,i)$ that is not in $V_i{}'$. We then create a BDD TB_j for $T(j)$ by conjuncting the BDDs of each of its parameter-value assignments. If $TB_j \wedge Valid = FALSE$, we iteratively call Algorithm 2 to locate a minimal invalid value combination in $T(j)$, and then randomly choose an entry from the combination and uninitialize it in $T(j)$, thus turning the combination into a valid one. We then check again whether the modified $T(j)$ is valid, and proceed until all minimal invalid value combination are removed from $T(j)$.

Algorithm 2 works as follows. First, it initializes two BDD arrays with prefixes and suffixes of $T(j)$, i.e., entry i in the *prefix* array contains the BDD representing the prefix of $T(j)$ up to entry $i-1$ and entry i in the *suffix* array contains the BDD representing the suffix of $T(j)$ starting from entry $|T(j)|-i$ (line 3). Next, for each i, it creates a BDD representing $T(j)$ excluding $T(j,i)$ (line 8). If the BDD is invalid, the algorithm proceeds recursively to find a minimal invalid combination in $T(j) \setminus T(j,i)$ (line 10). If no subsets are invalid, then $T(j)$ is a minimal invalid combination (line 13).

The algorithm is based on the observation that if a combination of size k is a minimal invalid one, then each of its k subsets of size $k-1$ are valid. Hence it is enough to check the validity of these k subsets. Note that since the algorithm is greedy, the found invalid combination is only locally minimal, and other smaller invalid combinations may exist in the test. These combinations will be found in subsequent calls to the algorithm, if they still remain in the test after removing values from the found combination.

The complexity of Algorithm 2 is $O(|T(j)|^2)$. Note that the BDD conjunctions are performed between tiny BDDs, hence constant time can be assumed for them. The complexity of $F_{REPCOMB}$ is $O(|T(j)|^3 \times |T|)$. As we will show in Section 5, in practice the running time of the repair is negligible.

```
input : An array of BDDs $TC$ representing the invalid test.
        The BDD $Valid$ of all valid tests.
1  if size($TC$) == 1 then return TC;
2  $prefix(0) \leftarrow TRUE$; $suffix(0) \leftarrow TRUE$;
3  for $i \in \{0, \ldots, \text{size}(TC)-2\}$ do
4      $prefix(i+1) \leftarrow prefix(i) \wedge TC(i)$;
5      $suffix(i+1) \leftarrow suffix(i) \wedge TC(\text{size}(TC)-1-i)$;
6  end
7  for $i \in \{0, \ldots, \text{size}(TC)-1\}$ do
8      $exclI = prefix(i) \wedge suffix(\text{size}(TC)-1-i)$;
9      if $(Valid \wedge exclI) = FALSE$ then
10         return findMinimal ($TC \setminus TC(i)$, Valid);
11     end
12 end
13 return $TC$
```
Algorithm 2: Find Minimal Invalid Combination

Modify implementation. To implement F_{MOD} we use seeding again (as in F_{EN} implementation), however, in this case, since all partial tests in the seed have to be extended to complete ones, the modification to Algorithm 1 is more complex. We describe it below.

In addition to the modification to line 1 as presented in the implementation of F_{EN}, we also maintain a set T_P of partial tests from the seed that have not been extended yet to complete ones. We initialize T_P with all partial tests that appear in T by adding after line 1 the line:

```
$T_P \leftarrow$ extractPartialTests($T$)
```

Then, in each iteration, as long as T_P is not empty, we conjunct the BDD *Collected* from which tests are chosen with T_P, to ensure that the chosen test will be an extension of a partial one from T_P. This is performed by adding after line 3 the lines:

```
if size($T_P$) > 0 then
    $Collected \leftarrow Collected \wedge$ buildBDD($T_P$)
end
```

Once a test is indeed chosen, we remove from T_P all the partial tests which it extends by adding after line 10 the lines:

```
if size($T_P$) > 0 then
    $chosenFromSeed \leftarrow$ getContained($T_P$, $chosen$)
    $T_P \leftarrow T_P \setminus chosenFromSeed$
end
```

We also record which values are added to each partial test, so that we can later mark them to the user as shown in Section 3. Once T_P is empty, the algorithm proceeds with enhancement.

Note that by defining two separate steps for fixing invalid tests, repair and modify, rather than defining a single replace operation, we can easily perform a reduction of the fixing problem to a seeding problem. This is because following the repair operation, the resulting partial test plan complies with the model constraints and can be used as a seed to a CT algorithm.

Selection implementation. To implement F_{SEL}, we use the ITSM algorithm from [3]. This algorithm first calculates the set C of t-way tuples covered by T, and then greedily iterates over the existing tests. In each iteration it chooses a test that adds the most uncovered t-way tuples out of C, and adds it to the solution set S. It stops when all tuples from C are covered. The complexity of the algorithm is $O(|S| \times |C| \times |T|)$. To reduce the constant factor, it avoids unnecessary calculations by remembering the number of uncovered tuples from previous iterations and lazily updating it, and performs efficient bit operations while counting uncovered combinations. To reduce the size of S, it prioritizes tests by giving higher weights to less frequent tuples.

The implementation of the five different co-evolution alternatives follows directly from Definition 4.8.

5 EVALUATION

We present an evaluation of our work in terms of the results of our co-evolution technique when applied to real-world model and test plan evolution. We then continue with a case study demonstrating the viability and necessity of our technique in achieving efficient co-evolution when using CT for test design.

5.1 Real-World Evidence

The research questions guiding our evaluation are:

RQ1 When the SUT and therefore the model evolve, to what extent can existing tests be reused as is and how much coverage will they provide?

RQ2 Are the co-evolution alternatives we suggest effective in reducing the required number of new tests and how do they affect the total test plan size?

RQ3 How much effort may be involved in modifying tests in the MES and MESW alternatives and what is the impact on their effectiveness?

RQ4 How does our co-evolution computation perform in terms of running time on real model and test plan versions in practice?

Table 2: Corpus model version and test plan sizes

Percentile	# params	Aver. # values per param	# constr.	Test plan size
5	5.00	2.28	0.00	9.00
25	8.00	2.90	7.00	21.00
50	11.00	4.10	15.00	49.50
75	21.25	5.33	26.25	113.25
95	60.25	10.30	168.50	411.75

To answer these questions, we applied our co-evolution technique to the evolution of a large corpus of real-world models and their test plans, which we have obtained from IBM.

5.1.1 Model Corpus Used and Setup. We applied our co-evolution technique to the evolution of 48 real-world industrial models, with 116 versions and 68 version commits. 32 models have 2 versions, 13 models have 3 versions, 2 models have 4 versions, and one model has 5 versions. Anonymized files of all model versions are available at http://smlab.cs.tau.ac.il/ctd/. According to IBM, the models were written by different CT engineers over a period of 9 years, and originate from 13 different domains: firmware, PaaS, IaaS, file system, operating system, database, storage, analytics, banking, telecom, security, networking, and software applications (email, document management, finance, etc.). The models also capture different levels of testing, such as function test, system test, etc. We did not select the models according to any criteria, and they represent all data available to us from IBM. All runs of our co-evolution computation were performed on a Windows 7 dual core machine with 1.9 and 2.5 GHz cores and 8 GB RAM. Only one core was used in our experiments. The BDD package used was JDD [19].

Table 2 presents the model and test plan sizes across the entire corpus in terms of number of parameters, average number of values per parameter, number of constraints, and the size of the test plan (as generated by the CT alternative). We report percentiles for each of these metrics (5%, 25%, 50%, 75%, and 95%). For example, 50% of the 116 model versions have 11 or more parameters. Only 5% of the model versions have 10.3 or more values per parameter on average. The percentiles show high variability in all aspects of model and test plan sizes. They also show that many real-world models and test plans are complex and large.

For evaluation purposes we treated each commit separately, i.e., for each model commit, we first generated the existing test plan by running CT on the previous model version, and then computed the five co-evolution alternatives for the resulting test plan and the model commit.

We used pairwise coverage requirements for most of the models, with the exception of several models where the engineers who originally created and used the model specified other requirements: in two cases the requirements were 2-way for a subset of the parameters and 1-way for the others, and in one case they were 3-way for a subset of the parameters and 2-way for the others.

5.1.2 Results. **RQ1** Table 3 shows the percentage of existing tests that were invalidated by model changes in our corpus, and the interaction coverage achieved by the ones that were still valid in the new model version. The results show that for 32% of the 68 commits, all tests are invalidated, and for 45% of the commits, more than 70% of the tests are invalidated. The results further show that for 69% of the commits, the existing tests achieve a coverage of less than 50% of the interactions in the new model version.

Table 3: Percentage of invalidated tests and coverage of the remaining tests in all commits.

Invalidated tests: % Invalid	% of commits
0	24
Up to 30	18
30 to 70	13
Over 70	13
100	32

Coverage of tests: % Coverage	% of commits
0	37
Up to 50	32
50 to 80	12
80 to 95	12
95 to 100	4
100	3

Table 4: Reduction in number of new tests of each alternative and increase in total test plan size, relative to the CT (from scratch) alternative

	Reduction in new tests				Increase in total size			
Percent.	EN	ES	MES	MESW	EN	ES	MES	MESW
5	0.00	0.00	0.06	0.00	1.00	1.00	0.96	0.95
25	0.00	0.00	0.46	0.00	1.00	1.00	1.00	1.00
50	0.10	0.10	0.75	0.23	1.05	1.01	1.06	1.00
75	0.35	0.35	0.88	0.72	1.21	1.08	1.12	1.05
95	0.90	0.90	1.00	0.98	1.79	1.34	1.31	1.14

To answer [RQ1], the results show that many of the existing tests are invalidated, and that the remaining ones achieve inadequate interaction coverage in new model versions. Thus, reuse as is is insufficient. This evidence motivates the development of test evolution operations such as modification and enhancement.

RQ2 Table 4 compares all alternatives to the CT (from scratch) alternative in terms of number of new tests and total test plan size, for each of the 68 commits. The CT alternative potentially represents the highest number of new tests and the lowest test plan size. Note that in CT from scratch there will typically be no reuse of old tests, because real-world models' test spaces are huge; there are slim chances the algorithm will produce one of the old tests.

Table 4 (left) presents for each of the four alternatives its ratio of reduction in the number of new tests compared to the number achieved by CT. Table 4 (right) presents for each of the other four alternatives its ratio of total test plan size to the size achieved by CT. As before, we report percentiles for each of these metrics.

The results show that the MES alternative achieves a very significant reduction in the number of new tests compared to the other alternatives. For example, the median reduction for MES is 75%, while it is only 10% for EN and ES, and 23% for MESW. On the other hand, the price MES pays in increase in total test plan size is rather low. For example, the median increase for MES is only 6%.

We also observe that MESW achieves the lowest overall increase in total test plan size, and that ES achieves a lower test plan size than EN. MESW performs better than MES because it discards invalidated tests: it reuses less tests than MES, so the CT algorithm has more freedom to optimize the final test plan size. Typically, EN and ES achieve the same number of new tests, because the selection building block removes only old tests, as they are the only ones it may eventually find to be redundant in the new model.

Table 5: Average number of values per test to change for MES and number of new parameter values to add for MES and MESW.

Percentile	aver. # changed values per test	# added values per test
5	1.00	0.00
25	1.00	0.00
50	1.80	2.00
75	3.60	8.25
95	6.88	23.30

Table 6: Running time for all alternatives on all model commits. Each cell represents the percentage of commits for which the co-evolution computation running times was within the time frame on the left-most column.

Time	CT	EN	ES	MES	MESW
$\leq 1sec$	87	84	79	81	79
$\leq 10sec$	10	15	15	13	16
$\leq 1min$	3	1	4	6	3
$\leq 1.5min$	0	0	2	0	2

To answer [RQ2], the results show that the MES alternative achieves a very significant reduction in the number of new tests compared to all other alternatives, while paying very little in increase in total size. In addition, MESW achieves the lowest total size increase, and is always better than ES in reduction in number of new tests. ES is better than EN in terms of increase in total test plan size.

RQ3 We estimate the effort of modification by reporting the average number of values to change per test (in MES) and the number of new values to add per test (in MES and in MESW). Table 5 presents these two metrics. On the left, we show the average number of values per test to change in the MES alternative, for each of the 53 commits where tests were invalidated. The average number was calculated out of all invalidated tests only[2]. On the right, we show the number of new values to add per existing test in MES and MESW, for each of the 68 commits. Note that this is a single number per commit, which is equal to the number of parameters added in the commit. As above, we report percentiles for each of these metrics.

The results show that a relatively low effort may be involved in modifying existing tests. In 50% of the commits, only up to 1.8 values per test on average need to be modified, and in 75% of the commits only up to 3.6. We note that this is only a weak proxi of the actual effort, as some parameters and/or values may require more effort to change than others, and the relative number of parameters whose values need to change out of all parameters may also affect the modification effort. See Table 2 for the number of model parameters across commits. For example, it shows that only 25% of the commits have 8 or less parameters (hence 75% have more than 8 parameters).

In addition, the number of new values to add to existing tests is rather small. In 50% of the commits, only up to 2 new values are added. We note that as opposed to changed values, which occur in MES only, new values should be added in all alternatives, since any new test plan must cover the functionality of the new parameters as well as their interaction with the old parameters. The difference is that in MES and MESW, values for new parameters may be added both to existing tests and to new tests, while in the other alternatives they are added in new tests only.

[2] Calculating the average value change size out of the entire test plan would further decrease it. See Figure 4 for the percentage of invalidated tests across commits.

To answer [RQ3], the results show that a rather low effort is involved in modifying existing tests in terms of number of changed values and number of added values. Hence, the MES and MESW approaches are effective in practice.

RQ4 Table 6 shows the running time achieved by all five alternatives on all 68 commits. For the vast majority of the commits, the running time was almost instantaneous (79%-87% in up to one second, 94%-99% in up to 10 seconds), and reached up to 1.5 minutes in only two commits. In addition, though overall CT and EN were slightly faster than the others, and ES slightly slower than the others, there were no significant differences between the alternatives. We note that repair time for the MES alternative was negligible: in all but one case it took less than 0.5 second, and in all cases it ranged between 1% and 10% of the overall MES running time.

To answer [RQ4], the results show that all alternatives achieve excellent and similar running times in practice. Since computation is efficient, there is no need for the engineer to a-priori select the co-evolution method; Instead, she can choose the best alternative from the test plans produced by the tool.

5.1.3 *Threats to Validity.*

Internal Our implementation of the co-evolution alternatives may not be free of bugs. We mitigated this by extensive testing. **External** First, the models and versions used in the analysis might not be representative of real-world model evolution. To mitigate this threat, we chose all real-world models and versions available to us from IBM, as described in Section 5.1.1. Second, the average number of values to change may not be a perfect proxi for the modification effort, as some parameters may be more difficult to change than others. We currently have no better means to estimate this effort on the large corpus.

5.2 Real-World Industrial Project

We report on a real-world industrial project in which our co-evolution solution was used as part of the deployment of CT for test design.

MaaS360 is a commercial security product. The agile development mode of MaaS360 dictates a release every 6 weeks with many new features. Several QA teams consisting of more than 100 engineers are responsible for testing different parts and aspects of MaaS360. To speed up test time, tests are automatically run, but generating the automation scripts is a manual effort. Hence there is a clear motivation to maximize reuse of tests and their scripts.

When the QA teams of MaaS360 started adopting CT in order to reduce their test plan sizes to a manageable size while achieving a better sense of their test coverage, they quickly stumbled upon the evolution challenge, due to their frequently evolving test spaces. Their goal was to continue optimizing their test plans also during evolution, while on the one hand keeping the existing tests untouched as much as possible, and on the other hand pushing into the test plan as many new changes as possible.

As they explored our co-evolution solution, it was obvious that running CT from scratch was not an option due to the huge automation effort wastage it will cause. They estimated that in the long run, the EN approach would end up with a large bulky unoptimized set of tests. They considered the ES alternative, however it did not demonstrate enough reuse of their existing tests. Thus, MESW and even more so MES were the clear middle ground approaches they were looking for. Specifically, for the majority of cases, a change in a single model parameter amounted to a change in only one or two lines in the automation script. Test Steps and expected results (oracles) were defined in the CT model based on parameter values, and were changed together with them. The team considered this to be a reasonable effort, especially given that not many values were invalidated and had to change between different test plan versions. Thus, the QA teams of MaaS360 adopted the modification-based alternatives. Moreover, they explicitly stated to us that these alternatives were necessary for them to continue using CT in their agile development process and achieve efficient test plan evolution.

We demonstrate the use of our co-evolution solution for testing of MaaS360 on one of the many models they created, which captures a specific function of MaaS360. One version of the model contained 50 parameters, 13 constraints, and 3 values per parameter on average. The first 10 parameters described different payload configurations. Each of the remaining parameters described specific settings related to one of the first 10 parameters. There were pairwise coverage requirements on the first 10 parameters, and 1-way requirements on the others. The initial test plan had 9 tests.

The next version of the model contained 65 parameters, 20 constraints, and 3 values per parameter on average. 4 payload configuration parameters were added along with their associated settings parameters. In addition, one value was removed from one of the original settings parameters. This removal invalidated 3 out of the 9 existing tests, and the other 6 tests achieved only 47.1% coverage in the second model version. The CT alternative resulted in 12 tests, all of them new. EN and ES resulted in 16 tests, 10 of them new. MESW resulted in 12 tests, similarly to CT, where all 6 existing tests were modified and 6 new tests were required. Lastly, MES resulted in only 9 tests, all of them existing modified tests; no new tests were required. The modification involved adding the new payload configurations and settings to the existing tests, and changing the single invalid value in the 3 invalidated tests. Clearly, the MES alternative provided the best tradeoff in this case.

6 RELATED WORK

We discuss related work on test plan evolution in general and in the context of CT in particular.

Different aspects of test plan evolution have been studied with regard to (Java and C++) code. Pinto et al. [33] investigated how test plans evolve, specifically how tests are added, removed, and modified in practice. Given two versions of a program and its test plan, their tool automatically computes differences in the behavior of the test plans on the two program versions, classifies the actual repairs performed between the versions, and computes the code coverage attained by the tests on the two program versions. Mirza-Aghaei et al. [31] focus on automating test plans updates. They identify scenarios that allow either to repair tests or to use tests to generate new ones, and propose algorithms that automatically repair and generate tests by adapting existing ones. Marinescu et al. [28] present a framework to analyse code, test, and code coverage evolution. Automated incremental repair of UI and web-based test suites was investigated in [12, 16], respectively. None of these works deals with evolution in the context of CT.

Qu et al. [34] empirically examined the effectiveness of CT on regression testing in evolving programs with multiple versions. Co-evolution of the test plan and the model is not discussed in [34].

Most recently, we have presented syntactic and semantic differencing for combinatorial models [37], using BDD-based algorithms. This work is limited to differencing between model versions. Co-evolution of the test plan and the model is not discussed.

Finally, seeding, suggested by Cohen et al. [7] as a means for the engineer to specify tests that must appear in the solution, is related to our work. Czerwonka [10] suggests the use of seeding in the context of evolution. We use a form of seeding, based on existing tests, in our enhancement and modification building blocks.

7 CONCLUSION AND FUTURE WORK

In this work we propose a first co-evolution approach for combinatorial models and test plans, accompanied with an efficient implementation. Our approach consists of five alternatives to evolve a test plan following changes in the model from which it was derived, all meeting the coverage requirements while considering tradeoffs between the total test plan size and number of new tests. To achieve efficient co-evolution, they are based on three building blocks: minimally modifying tests, enhancing them, and selecting from them. We implemented our technique within a commercial CT tool, and evaluated it on 48 real-world models with 116 versions, demonstrating the need for evolution techniques, and our excellent performance times. Our results show that using all three building blocks provides an effective middle ground approach that balances between the co-evolution tradeoffs. We further report on an industrial project which adopted our solution and found it necessary to enable the use of CT in an agile development process.

We suggest the following future research. First, consider code coverage and fault finding effectiveness in the assessment of each of the alternatives, provided that one has access to such data, as in [14]. Second, evaluate our approach to co-evolution on multiple versions over time, possibly with higher interaction levels, to examine the accumulated effect of our technique, if such exists. Third, incorporate user input about the expected effort to modify individual parameters into the algorithms, to prioritise the modification of parameters that are easier to change in the test implementations.

ACKNOWLEDGMENTS

We thank Sushovan Bhattacharya for his collaboration on the industrial project. We thank Muhammed Tibi for his contribution to our solution implementation. This research was done under the terms of a joint study agreement between IBM Corporation Inc (via the IBM Research Lab - Haifa) and Tel Aviv University.

REFERENCES

[1] K. Z. Bell. *Optimizing Effectiveness and Efficiency of Software Testing: A Hybrid Approach*. PhD thesis, North Carolina State University, 2006.
[2] K. Z. Bell and M. A. Vouk. On effectiveness of pairwise methodology for testing network-centric software. In *2005 International Conference on Information and Communication Technology*, pages 221–235, 2005.
[3] D. Blue, I. Segall, R. Tzoref-Brill, and A. Zlotnick. Interaction-based test-suite minimization. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, 2013.
[4] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. on Comp.*, 35(8):677–691, 1986.
[5] K. Burroughs, A. Jain, and R. Erickson. Improved quality of protocol testing through techniques of experimental design. In *SUPERCOMM/ICC*, pages 745–752, 1994.
[6] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.
[7] 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 Trans. on Softw. Eng.*, 23(7):437–444, 1997.
[8] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn. Constructing test suites for interaction testing. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 38–48, 2003.
[9] M. B. Cohen, J. Snyder, and G. Rothermel. Testing across configurations: implications for combinatorial testing. *SIGSOFT Softw. Eng. Notes*, 31(6):1–9, 2006.
[10] J. Czerwonka. Pairwise testing in the real world: Practical extensions to test-case scenarios. In *PNSQC*, 2006.
[11] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-Based Testing in Practice. In *ICSE*, pages 285–294, 1999.
[12] Z. Gao, Z. Chen, Y. Zou, and A. M. Memon. Sitar: Gui test script repair. *IEEE Transactions on Software Engineering*, 42(2):170–186, 2016.
[13] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering*, 16(1):61–102, 2011.
[14] L. S. Ghandehari, J. Czerwonka, Y. Lei, S. Shafiee, R. Kacker, and R. Kuhn. An empirical comparison of combinatorial and random testing. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pages 68–77, 2014.
[15] M. Grindal, B. Lindström, J. Offutt, and S. F. Andler. An evaluation of combination strategies for test case selection. *Empirical Softw. Eng.*, 11(4):583–611, 2006.
[16] M. Hammoudi, G. Rothermel, and A. Stocco. Waterfall: An incremental approach for repairing record-replay tests of web applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 751–762, 2016.
[17] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation.* Addison-Wesley Professional, 2010.
[18] IBM Functional Coverage Unified Solution (IBM FOCUS). http://researcher.watson.ibm.com/researcher/view_group.php?id=1871.
[19] JDD. http://javaddlib.sourceforge.net/jdd/.
[20] P. M. Kruse. Test oracles and test script generation in combinatorial testing. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 75–82, 2016.
[21] D. R. Kuhn, R. N. Kacker, and Y. Lei. *Introduction to Combinatorial Testing*. Chapman & Hall/CRC, 2013.
[22] D. R. Kuhn, I. D. Mendoza, R. N. Kacker, and Y. Lei. Combinatorial coverage measurement concepts and applications. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 352–361, 2013.
[23] D. R. Kuhn and M. J. Reilly. An investigation of the applicability of design of experiments to software testing. 27th NASA/IEEE Software Engineering Workshop, NASA Goddard Space Flight Center, 2002.
[24] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, Jr. Software fault interactions and implications for software testing. *IEEE Trans. Softw. Eng.*, 30(6):418–421, 2004.
[25] R. Kuhn, Y. Lei, and R. Kacker. Practical Combinatorial Testing: Beyond Pairwise. *IT Professional*, 10(3):19–23, 2008.
[26] Y. Lei and K.-C. Tai. In-parameter-order: A test generation strategy for pairwise testing. In *The 3rd IEEE International Symposium on High-Assurance Systems Engineering*, HASE '98, pages 254–261, 1998.
[27] S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment. In *ICCAD*, pages 6–9, 1988.
[28] P. D. Marinescu, P. Hosek, and C. Cadar. Covrig: a framework for the analysis of code, test, and coverage evolution in real software. In C. S. Pasareanu and D. Marinov, editors, *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 93–104. ACM, 2014.
[29] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, 2003.
[30] S. Minato. *Graph-Based Representations of Discrete Functions*, pages 1–28. Springer US, 1996.
[31] M. MirzaAghaei, F. Pastore, and M. Pezzè. Automatic test case evolution. *Softw. Test., Verif. Reliab.*, 24(5):386–411, 2014.
[32] Pairwise testing website. http://www.pairwise.org/tools.asp.
[33] L. S. Pinto, S. Sinha, and A. Orso. Understanding myths and realities of test-suite evolution. In W. Tracz, M. P. Robillard, and T. Bultan, editors, *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, page 33. ACM, 2012.
[34] X. Qu, M. B. Cohen, and K. M. Woolf. Combinatorial interaction regression testing: A study of test case generation and prioritization. In *ICSM*, pages 255–264, 2007.
[35] 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*, ISSTA '11, pages 254–264, 2011.
[36] G. Seroussi and N. H. Bshouty. Vector sets for exhaustive testing of logic circuits. *IEEE Transactions on Information Theory*, 34(3):513–522, 1988.
[37] R. Tzoref-Brill and S. Maoz. Syntactic and semantic differencing for combinatorial models of test designs. In S. Uchitel, A. Orso, and M. P. Robillard, editors, *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 621–631. IEEE / ACM, 2017.
[38] D. R. Wallace and D. R. Kuhn. Failure modes in medical device software: an analysis of 15 years of recall data. In *ACS/ IEEE International Conference on Computer Systems and Applications*, pages 301–311, 2001.
[39] A. W. Williams. Determination of test configurations for pair-wise interaction coverage. In *TestCom*, pages 59–74, 2000.
[40] P. Wojciak and R. Tzoref-Brill. System Level Combinatorial Testing in Practice – The Concurrent Maintenance Case Study. In *ICST*, pages 103–112, 2014.
[41] Z. Zhang and J. Zhang. Characterizing failure-causing parameter interactions by adaptive testing. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 331–341, 2011.