

Combinatorial Transition Testing in Dynamically Adaptive Systems

Pierre Martou
ICTEAM, UCLouvain
Louvain-la-Neuve, Belgium
pierre.martou@uclouvain.be

Kim Mens
ICTEAM, UCLouvain
Louvain-la-Neuve, Belgium
kim.mens@uclouvain.be

Benoît Duhoux
ICTEAM, UCLouvain
Louvain-la-Neuve, Belgium
benoit.duhoux@uclouvain.be

Axel Legay
ICTEAM, UCLouvain
Louvain-la-Neuve, Belgium
axel.legay@uclouvain.be

ABSTRACT

Due to the large number of possible interactions and transitions among features in dynamically adaptive systems, testing such systems poses significant challenges. To verify that such systems behave correctly, the technique of combinatorial interaction testing (CIT) can be used to create concise test suites covering all valid pairs of features of such systems. However, while CIT claims to find all errors caused by two features, we show that it does not cover certain errors occurring only for specific transitions between two features. To address this issue we study in depth the complementary technique of Combinatorial Transition Testing (CTT). From an initial generation algorithm that combines both interaction and transition coverage, we propose an optimised version that reduces the size of generated test suites by ~30%, reconfiguration cost by ~27% and drastically stabilises these results. After multiple generations, the standard deviation on the sizes of generated test suites is reduced by ~81%. Based on a comprehensive analysis over a large number of feature models, we also conclude that the size of CTT-generated test suites is linearly correlated to CIT-generated ones and that combinatorial transition testing also grows logarithmically in the number of features.

CCS CONCEPTS

• **Software and its engineering** → **Real-time systems software**; *Feature interaction*; **Software testing and debugging**; *Software product lines*.

KEYWORDS

combinatorial testing; transition testing; dynamically adaptive software systems; dynamic software product lines; feature modelling; software testing

ACM Reference Format:

Pierre Martou, Benoît Duhoux, Kim Mens, and Axel Legay. 2024. Combinatorial Transition Testing in Dynamically Adaptive Systems. In *18th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2024)*, February 7–9, 2024, Bern, Switzerland. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3634713.3634726>

1 INTRODUCTION

Dynamically adaptive software systems are systems that adapt their behaviour dynamically to their surrounding environment. In this paper, we assume that the behaviour of such systems is described in terms of features organised in a feature model. Akin to software product lines, such systems can thus be seen as an implementation of a family of systems whose actual configuration (selection of active features) depends on the current state of the environment. Even though such feature models are highly constrained, they may still generate an overwhelming number of potential configurations, making it infeasible to test all of them exhaustively.

A dedicated testing technique is therefore needed to generate a concise yet representative test suite covering all valid combinations of features. The popular technique of Combinatorial Interaction Testing (CIT) can achieve exactly this. With sufficiently high degrees of coverage, where n -wise interaction testing means that all combinations of size n will be covered, it can find most errors in a large variety of systems [17, 22]. Furthermore, recent research has applied CIT to feature-based context-oriented systems [19], a particular class of dynamically adaptive software systems [8].

A benefit of CIT is that it produces only relevant configurations to be tested. Once test configurations have been generated, a good test oracle is still needed to determine whether the system behaves as expected for each configuration. Assuming a perfect test oracle, if a given test configuration contains a combination of features that may together provoke erratic behaviour due to a faulty interaction of these features, the oracle will indeed detect this error or fault. We will refer to such errors caused by an incorrect interaction between specific features as *behavioural interaction errors*. Using a perfect oracle, n -wise CIT is guaranteed to detect all behavioural interaction errors (corresponding to interactions of size n or less). The contributions of this paper are captured by the three research questions presented below.

RQ1 Which behavioural transition errors can CIT detect?

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

VaMoS 2024, February 7–9, 2024, Bern, Switzerland

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0877-0/24/02...\$15.00

<https://doi.org/10.1145/3634713.3634726>

In a recent workshop paper [18] it was argued that CIT may miss certain relevant scenarios that exhibit certain kinds of errors that occur only for particular feature transitions. These errors are not caused by the simultaneous presence of a set of features, but rather by changes caused by the simultaneous activation or deactivation (called transitions) of a set of features. Switching the activation state of a specific set of features may trigger *behavioural transition errors* in the resulting configuration. To generate test suites that capture such transition errors as well, the paper theorised (i.e. without any proof-of-concept, algorithm or validation) a new alternative technique called Combinatorial Transition Testing (CTT). That paper thus provided a preliminary answer to RQ1. In this paper, we further answer RQ1 and argue that CIT cannot detect behavioural transition errors consistently, e.g. by showing that the transition coverage of CIT testing approaches is incomplete.

RQ2 *How can we generate a test suite that covers all behavioural interaction and behavioural transition errors?*

In the current paper, we apply Combinatorial Transition testing to the class of dynamically adaptive software systems. We propose an initial (greedy) generation algorithm that validates the pertinence and feasibility of CTT, then compare it with an improved version that includes three optimisations to enhance the algorithm's efficiency. These optimisations reduce by ~30% on average the size of generated test suite, while keeping full transition coverage.

Note that detecting behavioural transition errors is not relevant for non-dynamically adaptive systems. Since configurations chosen for instances of those systems do not change dynamically, behavioural transitions typically do not occur.

RQ3 *What is the test effort of adding transition coverage to interaction coverage?*

We conduct a quantitative study to compare the coverage and test efforts between CIT and CTT. For this we use the public feature model repository S.P.L.O.T. [20]. We confirm that CTT is linearly correlated to CIT with a factor of approx. 2, and hence has logarithmic growth in the number of features.

This paper is structured as follows. After discussing related work in Section 2, Section 3 presents an example of a dynamically adaptive software system. Section 4 shows that a testing approach based on CIT is not able to consistently find *behavioural transition errors* in such systems. Section 5 introduces the main concepts of our *Combinatorial Transition Testing* (CTT) approach. Section 6 then presents the optimised greedy algorithm. Section 7 analyses the performance of these optimisations, and compares CTT to standard CIT. Section 8 discusses some threats to the validity of our findings. Section 9 concludes and discusses remaining challenges to fully turn CTT into a single comprehensive testing technique.

2 BACKGROUND AND RELATED WORK

Let us first describe the vocabulary used in this paper, inspired by Combinatorial Interaction Testing and feature modelling. Given a feature f , a function $v : f \rightarrow \{\text{activated}, \text{deactivated}\}$ assigns a specific value to a feature f . A partial test configuration is defined as a set of value assignments $\{v_1, \dots, v_n\}$ for a set of features $\{f_1, \dots, f_n\}$. To simplify the notation, we will write $+f_i$ if the value *activated* is assigned to f_i and $-f_i$ when its value is *deactivated*. A partial

configuration $\{-\text{High}, -\text{Map}\}$ thus represents the fact that the features *High* and *Map* are both *deactivated*. A partial configuration is valid when it represents an assignment that satisfies all constraints imposed by the feature model.

The goal of Combinatorial Interaction Testing (CIT) is to cover all valid partial configurations (or interactions) $\{v_1, \dots, v_n\}$ of size n . A test configuration in a test suite *covers* an interaction if this configuration contains all value assignments of the interaction. Consider three features *High*, *Low* and *Map* without constraints, the test configuration $\{+\text{High}, -\text{Low}, +\text{Map}\}$ covers the interactions $\{+\text{High}, -\text{Low}\}$ and $\{+\text{High}, +\text{Map}\}$ of size 2, among others.

The goal of n -CIT is to cover and find all errors related to the interaction of n features. In this paper we focus on 2-way interactions only. Going beyond CIT, in this paper we are interested in Combinatorial Transition Testing (CTT). Whereas we will define this concept more formally in section 5, intuitively, with transition testing we want to check errors due to transitions between configurations, i.e. errors due to a change of value assignments of some features from one configuration to another. Such transitions are particularly relevant and frequent in dynamically adaptive systems.

An alternative approach to cover all transitions is that of State Transition Testing [6]. As far as we know, state transition testing is limited to systems which explicitly represent transitions between states (e.g., using Petri Nets [24]). Such techniques rely on exploring the entire graph, i.e. all existing edges (possible transitions) between all nodes (possible system states).

In this paper instead we investigate transition testing on systems designed with feature modelling, with no explicit description of possible transitions. Even if we could generate a connected graph from a feature model, exploring all transitions would not be feasible in practice. As a result our CTT approach derived from CIT will propose its own transition testing definition and coverage criterion, based on feature modelling.

Combinatorial Sequence Testing [11, 16] is another alternative approach that covers all sequences of k specific events in a system. Again, the impracticality of this approach for dynamically adaptive software systems arises from the exponential number of configurations ($\approx$ events) they can generate. For a single configuration, however, this approach could be used to verify different activation orders of features ($\approx$ events). This could be very useful for dynamically adaptive software systems whose feature activation order yields inconsistencies, but this is out of scope of the current paper.

3 RUNNING EXAMPLE

To illustrate that CIT does not find all behavioural errors that our approach could, we describe a dynamically adaptive system used as running example. We then detail three behavioural errors.

3.1 Risk information system

Our running example is that of a Risk Information System (RIS). Such a system can inform users when calamities occur and provide them with specific instructions to follow in such situations. For the sake of simplicity, our running example supports only two types of hazards: cold weather and floods, but can be generalised to handle other types of hazards such as earthquakes or a pandemic. Figure 1

illustrates the system’s feature model, designed according to the FODA notation [14].

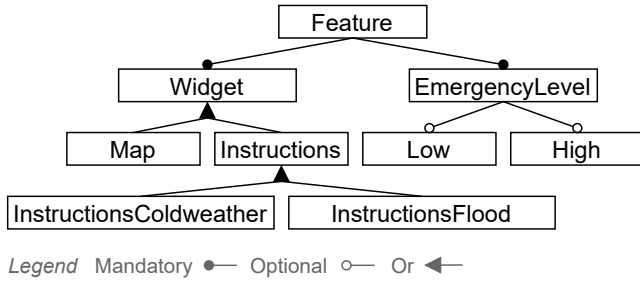


Figure 1: Feature model of a simplified version of the Risk Information System.

The main features in this model are *Widget* and *EmergencyLevel*. The *Widget* feature determines what graphical information is shown to users, such as a *Map* or safety *Instructions* on a specific hazard. The *EmergencyLevel* feature determines the emergency level of an ongoing hazard which may currently affect the user. Two different levels are distinguished, *Low* and *High*, depending on the severity of the hazard. As we consider a cold weather hazard to be less dangerous than a flood, their emergency levels are low and high, respectively. For simplicity in the prototype system, let us assume that the *EmergencyLevel* is implemented as a mere integer variable. A value 0 means that there is no ongoing hazard. The variable is set to 1 when an emergency of low level occurs, and is set to 2 if it has a high emergency level. When multiple hazards with different emergency levels occur simultaneously, the highest emergency level takes precedence.

3.2 Behavioural errors

Now that we have set the context of our running example, we present how an incorrect implementation of the system could cause the system to behave erroneously. For conciseness, we only detail with a code snippet the first error, but a complete implementation is available in the following repository: <https://github.com/PierreMartou/VaMos2024-Prototype>.

Randomness in test results related to non-determinism or flakiness [10] could arise when examining transitions between two configurations if they contain multiple feature (de)activations at the same time. To circumvent this issue we use a deterministic (de)activation order inside a transition. In this paper, we first deactivate the features and then activate features. Because feature models have hierarchical (parent-child) relations between features, we also ensure that parent features are always activated before their children, and deactivated after them.

A first behavioural transition error (**T1**) causes a wrong emergency level when the system deactivates *High* and activates *Low* at the same time. This transition is defined as transition $|-High, +Low|$. This behavioural transition error is explained below by following the implementation of the deactivation of the *High* feature and the activation of the *Low* feature. Assume an initial configuration $\{+High, -Low\}$ and a correct emergency level set to the value of 2. On transition $|-High, +Low|$, the system first deactivates *High*

and then activates *Low*. When *High* is deactivated, its corresponding implementation (Listing 1) is executed. As *Low* is in the final configuration, the emergency level is set to the value of 1. This deactivation is then followed by the activation of *Low* and its corresponding implementation (Listing 2) is executed in turn. As the emergency level is less than 2, this variable is incremented by 1, resulting in an unintended emergency level 2 (instead of 1).

```

1 def high_deactivation():
2     if {+Low} in final_config:
3         emergency_level = 1
4     else:
5         emergency_level = 0
  
```

Listing 1: Implementation of deactivating the *High* emergency level feature.

```

1 def low_activation():
2     # if emergency_level == 0:
3     if emergency_level < 2:
4         emergency_level += 1
  
```

Listing 2: (Erroneous) implementation of activating the *Low* emergency level feature. (Possible fix in commented line 2.)

Observe that reaching this final configuration through any other transitions does not result in an error. The emergency level will be set to 1 if we come from the initial configuration $\{+Low, +High\}$ to the same final configuration $\{+Low, -High\}$, for example. In addition, by sequencing the deactivation of *High* and activation of *Low* from the same initial configuration $\{-Low, +High\}$ to reach the same final configuration $\{+Low, -High\}$ through the intermediate configuration $\{-Low, -High\}$, the result of the emergency level will also be set to 1. This means that this behavioural transition error only occurs when both are in the same transition.

The application implements two widgets, *Map* and *Instructions*. When only one is active, it should be on the topmost position in the user interface. However, another behavioural transition error (**T2**) results in an empty space appearing at the top of the user interface, when the *Map* feature is deactivated and the *Instructions* feature is activated simultaneously, defined as transition $|-Map, +Instructions|$.

The last error is a behavioural interaction error (**I3**) between *InstructionsColdWeather* and *InstructionsFlood*, defined as interaction $\{+InstructionsColdWeather, +InstructionsFlood\}$. Their execution results in the flood-specific instructions overwriting the instructions dedicated to cold weather, instead of showing the two sets of instructions one after the other.

While it could be argued that such errors stem from poor design, and could have been avoided by eliminating the transitions that trigger behavioural errors, this misses the essential point. The necessity for a dedicated testing technique arises because of the potential for such errors. Redesigning our case study to eliminate the error-inducing transitions is indeed a solution, but the critical question is which testing approach effectively ensures the absence

of transitions that could lead to errors. It is important to acknowledge that if a feature model has valid transitions (i.e., it has at least two valid configurations), then it inherently possesses the potential for behavioural transition errors.

4 PERTINENCE OF CIT

The three behavioural errors of our running example are caused by undesired interactions between two features. This leads to the first research question (RQ1): which behavioural transition errors can CIT detect? CIT is a well-known technique to generate test suites whose size grows logarithmically in the number of features, hence preventing combinatorial explosion [4, 5]. While Combinatorial Interaction Testing (CIT) is a popular technique to find interaction errors, we argue that it cannot consistently discover all *behavioural transition errors* (such as **T1** and **T2**) in dynamically adaptive systems. To demonstrate this claim, we generate a test suite using CIT to cover all pairs of features of our running example, i.e. a covering array.

Test suites generated by CIT are usually described by listing their test configurations. For fair comparison, we assume that these test configurations are executed sequentially as executing them in all possible orders would result in combinatorial explosion. To better study transition errors that only occur in transitions *between* configurations, we represent the test suite with only the set of feature (de)activations to reach the next configuration (i.e. transitions). This test suite is depicted in Table 1. Assume the following scenario which starts from an empty initial configuration. The user launches the application and senses cold weather. In this configuration, the system has to activate the features *Widget*, *Instructions*, *InstructionsColdWeather*, *EmergencyLevel*, and *Low* (test 1). A few days later, the weather conditions change. The user sees that the cold weather is over but that it rains a lot. This causes the area to be flooded. To adapt to its new environment, the system deactivates the features *InstructionsColdWeather* and *Low*, and activates the features *InstructionsFlood* and *High* (test 2).

Table 1: Test suite that covers all interactions, in terms of transitions. *Instructions* is shortened to *Instr*.

Test	Features
0	(No features active.)
1	+Widget, +Instr, +InstrColdweather, +EmergencyLevel, +Low
2	-InstrColdweather, -Low, +InstrFlood, +High
3	-Instr, -InstrFlood, +Map
4	+Instr, +InstrFlood, +InstrColdweather, +Low
5	-InstrFlood, -High
6	-Instr, -InstrColdweather, -Low
7	+Low
8	+High

In this test suite we observe that the behavioural interaction error **I3** associated to the interaction {+InstructionsColdweather, +InstructionsFlood} is covered by test 4. However it does not cover the transitions |-High, +Low| nor |-Map, +Instructions|. Hence the behavioural transition errors (**T1** and **T2**) will remain unnoticed. Although this test suite covers all feature interactions, finding all behavioural transition errors appears to be a much more difficult

task. Note that the test suite does activate *Low* in test 4 and deactivates *High* in test 5. However, this does *not* allow the error **T1** to get noticed since for this both must be part of a single transition. Regarding error **T2**, deactivating *Map* simply never happens, let alone deactivating *Map* and activating *Instructions* simultaneously.

Even though CIT was not designed to detect all behavioural transition errors, we want to assess its efficiency of discovering such errors. Therefore we rely on a greedy generation algorithm which was already applied with success to feature-based context-oriented systems, a particular class of dynamically adaptive systems, to generate test suites [5, 19]. To reduce the impact of the inherent randomness of greedy approaches, we generated 1000 test suites. After generating these test suites, we compute the number of times the transition |-High, +Low| (**T1**), the transition |-Map, +Instructions| (**T2**), both (**T1&T2**), or all transitions (**All**) are present in these test suites. Table 2 presents these results in percentage (i.e. the chance to find a specific transition in a generated test suite). These percentages are calculated following four distinct rearrangement strategies that modify the order of the configurations. In practice, reordering is widely used to optimise the application of generated test suites [25]. The *Unordered* strategy involves no reordering. The *Random* strategy involves shuffling the configurations randomly. The *Dissimilarity* strategy aims to maximise differences between successive test configurations, with the hypothesis that quickly exploring very different configurations would discover errors faster [3, 23]. The *Cost* strategy minimises the simulation effort or reconfiguration cost, which is defined as the number of activations and deactivations required to transition between all test configurations [19].

Table 2: Percentage of test suites containing the errors T1, T2, both, and All transitions, with different rearrangements.

	T1	T2	T1 & T2	All
Unordered	58%	92%	52%	0%
Random	42%	73%	30%	0%
Dissimilarity	50%	99%	49%	0%
Cost	36%	22%	8%	0%

The results reveal that strategies *Unordered* and *Dissimilarity* have a higher probability to discover errors **T1** and **T2** than the two other strategies *Random* and *Cost*. As the *Dissimilarity* strategy prioritises tests that differ the most, test suites contain far more transitions between test configurations and hence achieve a higher score, which was to be expected.

However, note that this strategy is not the most appropriate for dynamically adaptive systems because it causes a much higher simulation effort (reconfiguration cost) than the **Cost** strategy [19] (around twice more). Increasing this cost can lead to different issues: longer execution time due to more (un)deployment of features, more difficulties in locating a specific error due to larger transitions between scenarios (especially for behavioural transition errors), worse maintenance of the test suite, more difficult maintenance between system versions, etc. In Section 7, we will use such metrics to assess the performance of our testing approach.

Observe that the probability of detecting *both* errors (**T1** and **T2**) is the product of the probabilities of finding **T1** and **T2**. Because of

this probability rule, the probability of covering *all* valid transitions in a single test suite is close to zero, no matter the reordering.

We also compute the transition coverage, i.e. the number of transitions covered in a test suite divided by the total number of valid transitions¹. The average **transition coverage** for all rearrangement strategies is presented in Table 3. Additionally, let us assume that we generate multiple test suites and combine them until we achieve maximum transition coverage (100%). The average number of generated test suites to achieve this maximum coverage is shown in column **#suites for 100%**. Standard deviation for all measures are added in parentheses.

Table 3: Analysis of transition coverage with different rearrangements, averaged over 1000 test suites.

	Transition coverage	#suites for 100%
Unordered	60% (± 8)	11 (± 2)
Random	44% (± 11)	15 (± 4)
Dissimilarity	63% (± 8)	16 (± 6)
Cost	19% (± 5)	189 (± 116)

Even using the *Dissimilarity* strategy which is expected to perform the best, only $\sim 63\%$ of all transitions are covered in a test suite, on average. And the standard deviation shows that we would be extremely lucky to achieve over 71% of coverage.

Finally, **#suites for 100%** shows that generating multiple test suites to achieve full transition coverage is not a viable strategy. The problem is that because the transition coverage of the generated test suites largely overlap with each other, at least 11 test suites would be needed to achieve full coverage (for the *Unordered* case, and 16 on average for the *Dissimilarity* strategy). This overlapping is significantly intensified for the **Cost** strategy because of biases: no matter the test suite, some specific transitions are more optimal than others to reduce overall cost and the strategy often makes them appear, which results in enormous overlap.

5 COMBINATORIAL TRANSITION TESTING

As demonstrated in the previous section, CIT does not find all behavioural transition errors. Therefore we suggest an alternative testing approach, Combinatorial Transition Testing (CTT), whose goal is to obtain total transition coverage. The purpose of this section is to formalise transition testing and explain its similarities and differences with CIT, setting the groundwork for the introduction of the algorithmic solution given in Section 6.

In the same way as CIT, *n*-wise *combinatorial transition testing* should generate a test suite covering all *transitions* of size *n*. Note that a transition is also described with value assignments but refers to a *change* between configurations instead. The main difference of transition testing lies in a new definition of transition coverage:

DEFINITION 1. *We say that a test suite covers a transition $|v_1, \dots, v_n|$ if, in two consecutive configurations, the second configuration includes the assignment of values $\{v_1, \dots, v_n\}$ while the first configuration includes the opposite assignment, i.e., $\{-v_1, \dots, -v_n\}$.*

¹Valid transitions are formally defined in the next section, Definition 2.

For example, the transition $|+High, -Low|$ is covered by a test suite having the two successive configurations $\{-High, +Low, +Map\}$ and $\{+High, -Low, +Map\}$. From this example, we can also observe that the first and second configuration must be valid for the transition to be possible. We generalise this with the following definition:

DEFINITION 2. *We say that a transition $|v_1, \dots, v_n|$ is valid if the specified value assignment $\{v_1, \dots, v_n\}$ and its opposite $\{-v_1, \dots, -v_n\}$ satisfy all system constraints (i.e. both value assignments are valid).*

Finally we analyse how transition testing and interaction testing are related. It is interesting to note that if a test suite covers the transition $|+High, -Low|$, then it necessarily covers the associated interaction $\{+High, -Low\}$. The last definition extends this relationship to a broader context:

DEFINITION 3. *Covering the transition $|v_1, \dots, v_n|$ implies covering the corresponding interaction $\{v_1, \dots, v_n\}$. These interactions are called invertible interactions. Interactions whose opposite is invalid have no corresponding transition (non-invertible interactions), from Definition 2. Interactions are either invertible or not.*

An interesting property follows from these definitions. We can observe from Definition 3 that a test suite that covers all transitions does not necessarily cover all interactions, just like interaction testing does not cover all transitions. Therefore combining both transition and interaction coverage is a necessity in an ideal testing approach. Section 6 will thus present a generation algorithm that follows this approach.

To investigate deeper how transitions and interactions are connected, we compare the number of transitions ($\#transitions_n$) to the number of interactions ($\#interactions_n$) of size *n*. To this end, the number of invertible ($\#invert_int_n$) and non-invertible interactions ($\#non_inv_int_n$) is used.

$$\#invert_int_n = \#transitions_n \quad (1)$$

$$\#invert_int_n + \#non_inv_int_n = \#interactions_n \quad (2)$$

Equation 1 and 2 follow directly from Definition 3. We can then derive Equation 3 from the previous two.

$$\#transitions_n + \#non_inv_int_n = \#interactions_n \quad (3)$$

Equation 3 demonstrates that all interactions can be covered in an algorithm covering all transitions (i.e. invertible interactions) and all non-invertible interactions. Crucially, this algorithm will strive to cover the same number of tuples as typical CIT algorithms, which have already been proved to scale to a high degree using different techniques [2].

Care should be taken to avoid confusion regarding these equations, particularly Equation (1). It is essential to note that, even if the number of transitions and invertible interactions is the same, their coverage criteria differ. Coverage of invertible interactions does not imply coverage of transitions. Equation (3) will form the basis for the initial optimisation discussed in Section 6.2.

6 GREEDY GENERATION APPROACH

Now that we have formalised transition testing, we are ready to answer research question (RQ2): how can we generate a test suite that covers all behavioural interaction and behavioural transition errors? One possible answer could be to generate dozens of test

suites through CIT until all transitions are covered. However, this would result in an enormous test effort, which is far from ideal in practice. Instead, we propose and optimise a generation algorithm that covers both interaction and transition behavioural errors more efficiently.

The foundation of the algorithm is a row-by-row greedy algorithm for highly-constrained systems [5]. We improved it to add transition coverage. The base algorithm is presented in Section 6.1, the different optimisations are introduced in Sections 6.2 through 6.4, and the fully optimised version is described in Section 6.5.

The three main improvements to the base algorithm greatly reduce the size of generated test suites and stabilise the algorithm’s performance. The optimisations consist of filtering invertible interactions (Improvement 1, Section 6.2), adding a look-ahead mechanism (Improvement 2, Section 6.3) and proposing a comparative mechanism (Improvement 3, Section 6.4).

6.1 Generation algorithm

Consider k features f_i in a system, an empty list of test scenarios, a list of *uncovered* valid interactions, and a list of *uncovered* valid transitions. Valid *interactions* are computed by listing all valid (feature, value) combinations, i.e. that satisfy all constraints in SAT solving, and whose opposite is invalid. Valid *transitions* are generated by listing all (feature, value) combinations, while ensuring that both this combination and its opposite are satisfied with SAT solving (see Definition 2).

Each new test scenario added to the final covering array is the best candidate among M (M is an input parameter to the algorithm, usually set between 30 and 50), the best being the one with the highest score. The score of a candidate scenario is the number of uncovered interactions and transitions it covers. The generation of a candidate has the following strategy:

- (1) Compute the *score* of each possible value (i.e. activated or deactivated) for all features. The score of a value is given by the number of uncovered interactions and transitions that contain it. Following Definition 1, a transition can be covered only if its opposite value assignment is included in the previous test case. Only those uncovered transitions are considered. Find the value v associated to the highest score and add it to the new test scenario. Break ties randomly.
- (2) Once the first feature f_1 has been decided, assign all remaining features randomly to $f_2, \dots, f_k$. Expanding the search scope through randomness allows the generation algorithm to discover optimal paths and is exploited in Step 3.
- (3) Consider a valid partial assignment where all features $f_1, \dots, f_{n-1}$ have been assigned to $v_1, \dots, v_{n-1}$. Now, compute the score for both values of the feature f_n and choose the highest. The score of a value is given by the number of uncovered interactions and transitions that would be covered by this test scenario with this value. If adding a particular value makes the partial assignment invalid, then the other one is chosen by default. (The validity of a partial assignment is verified through SAT solving). Break ties randomly.

Note that, as changing the order of test scenarios modifies the transitions they cover, the test suite cannot be rearranged. This challenge is further discussed in Section 9.

6.2 Improvement 1: Invertible interactions filter

The main insight to all improvements is illustrated in Figure 2 based on our case study. It depicts the evolution of interaction and transition coverage of a test suite for the baseline algorithm (i.e. without improvements). The line stopping means that the coverage reaches 100%.

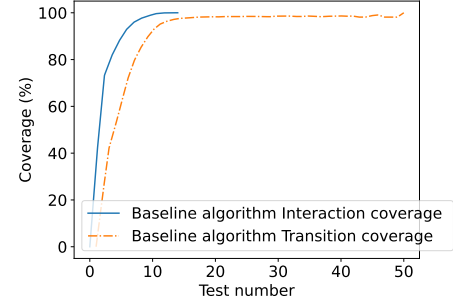


Figure 2: Evolution of the interaction/transition coverage of a test suite (averaged over 1000 test suites)

The transition coverage lacking behind interaction coverage is a clear bottleneck. This imbalance can intuitively be explained: contrary to transitions, interactions do not depend on the previous test case, which makes them easier to cover. Although this insight will be the foundation of our improvements, this tendency cannot be corrected. As covering a transition means covering its corresponding interaction, the evolution of transition coverage can at best be at the same level as the evolution of interaction coverage, but never overtake it. The goal of our optimisations is therefore to prioritise and prepare for optimal transition coverage.

The first improvement builds upon Definition 3 and Equation 3 to reduce the bias towards interaction coverage. These state that a test suite that covers a transition T necessarily covers its corresponding interaction I . In practice, this implies means that trying to cover an invertible interaction I early in the test suite is sub-optimal, as it will anyway be covered when its corresponding transition T is covered. These invertible interactions can thus be deleted from the list of uncovered interactions *before* starting the generation of test scenarios, which will drastically reduce their impact when computing scores at step 1 and 3 of the algorithm. This optimisation can also alleviate computations since it removes redundancy.

6.3 Improvement 2: Look-ahead mechanism

Another shortcoming can be observed in Figure 2: the algorithm reaches a plateau in transition coverage after around 14 test configurations. Indeed, the generation of test scenarios only ends when all transitions are covered. If only one transition is left uncovered but the latest test scenario does not allow its coverage (by Definition 1), then the algorithm will rely on its randomness to expand its search scope, which means it will generate new test scenarios until it eventually generates one that covers the last transition.

To solve the plateau issue and further prioritise transition coverage, we propose a *look-ahead* mechanism. It gives weight to interesting values that prepare for *future* transitions through a

look-ahead weight $w_{look-ahead}$. In our example, if the transition $[+High, -Low]$ is uncovered and the previous test scenario does not allow its coverage, then the values $\{-High\}$ and $\{+Low\}$ will be given a score of $w_{look-ahead}$ in both step 1 and step 3 of the generation algorithm.

Note that a look-ahead weight is only given to uncovered transitions whose coverage is prevented by the latest test scenario. Let $\#(uncovered_{f,v})$ be the number of uncovered interactions and transitions that contain the value v for feature f , $\#(uncoverable_{tf,v})$ the number of uncovered transitions that contain the value v for feature f and that cannot be currently covered, then the weighted look-ahead score $s_{laf,v}$ is:

$$s_{laf,v} = \#(uncovered_{f,v}) + w_{look-ahead} * \#(uncoverable_{tf,v}) \quad (4)$$

The look-ahead weight should be less than 1, if not the greedy algorithm would prefer preparing future transitions instead of covering actual transitions. Consequently, a value of 0 and 1 for the look-ahead weight actually generates equivalently bad results (i.e. high number of test scenarios). We observed that a weight between 0.4 and 0.6 produces the best results, with 0.5 consistently being the best value. For our case study, it reduces the average size of test suites from 18 to 13. This value is a very intuitive trade-off: covering a transition *now* is equivalent to preparing two *future* transitions.

6.4 Improvement 3: Comparative mechanism

In the same vain of better directing the algorithm to avoid non-optimal paths, the third improvement modifies step 1 of the algorithm which decides the first value of a new test scenario and hence its main direction. Indeed, the baseline step 1 blindly chooses the best score, while it could preemptively prevent a bad score from being chosen. As we know that all features will either be active or deactivated, this improvement will instead weigh the score associated to active and deactivated against each other. It means that even if a feature has a very high value for its active status, if its inactive status is equally high, then we will not prioritise choosing this value in step 1 as the very first value. Thanks to the comparative mechanism, the algorithm may instead choose a moderately high score but whose opposite is very low, and hence prevent this opposite score to be chosen further the line in step 3.

More generally, if $s_{laf,v}$ is the initial score (already weighted with the look-ahead heuristic) for a feature f and value v , then its comparatively-weighted score $s_{comp_{f,v}}$ is :

$$s_{comp_{f,v}} = s_{laf,v} - w_{comparative} * s_{laf,-v} \quad (5)$$

The comparative weight should be lower than 1 to avoid non-ideal paths, but we still want to build new test scenarios around values with high coverage potential. It is highly preferable to build a test scenario around a feature value with very high potential even if its opposite value also has high potential (e.g. a score of 1000 and 500 respectively), instead of a low-potential value whose opposite is even lower (e.g. a score of 501 and 0). We experimentally found that a weight of 0.3 consistently finds the best results.

6.5 Optimised generation algorithm

Consider k features f_i in a system, an empty list of test scenarios, a list of *uncovered* valid **non-invertible interactions** (**Improvement 1**), and a list of *uncovered* valid transitions.

Valid **non-invertible interactions** are computed by listing all valid (feature, value) combinations, i.e. that satisfy all constraints in SAT solving, and whose opposite is invalid. Valid *transitions* are generated by listing all (feature, value) combinations, while ensuring that both this combination and its opposite are satisfied with SAT solving (see Definition 2).

A look-ahead mechanism uses a weight $w_{look-ahead}$ and a comparative mechanism improvement uses $w_{comparative}$ as weight, both parameters of the algorithm. We recommend using a value of 0.5 and 0.3, respectively, as justified previously.

Each new test scenario added to the final covering array is the best candidate among M (M is an input parameter to the algorithm, usually set between 30 and 50), the best being the one with the highest score. The score of a candidate scenario is the number of uncovered interactions and transitions it covers, **plus the number of uncovered transitions that could possibly be covered in the next scenario, weighted by $w_{look-ahead}$** (**Improvement 2**). The generation of a candidate has the following strategy:

- (1) Compute the *score* of each possible value (i.e. activated or deactivated) for all features. The score of a value is $s_{comp_{f,v}}$, described in Equation 5 (**Improvement 3**), which will balance the score through both the look-ahead and comparative mechanism. Following Definition 1, a transition can be covered only if its opposite value assignment is included in the previous test case. Only those uncovered transitions are considered. Find the value v associated to the highest score and add it to the new test scenario. Break ties randomly.
- (2) Once the first feature f_1 has been decided, assign all remaining features randomly to $f_2, \dots, f_k$. Expanding the search scope through randomness allows the generation algorithm to discover optimal paths and is exploited in Step 3.
- (3) Consider a valid partial assignment where all features $f_1, \dots, f_{n-1}$ have been assigned to $v_1, \dots, v_{n-1}$. Now, compute the score for both values of the feature f_n and choose the highest. The score of a value is described in Equation 4 (**Improvement 2**), which will use the look-ahead mechanism to adjust the score of each value. If adding a particular value makes the partial assignment invalid, then the other one is chosen by default. (The validity of a partial assignment is verified through SAT solving). Break ties randomly.

7 VALIDATION

The previous section proposed a test generation algorithm that combines both interaction and transition testing. The final research question of this paper (RQ3) is: what is the test effort of adding transition coverage to interaction coverage? We start by analysing the impact on test effort of the three improvements of our algorithm on our illustrative case study. We then generalise by using the S.P.L.O.T. [20] repository (Software Product Lines Online Tools) and compare the performance of both CIT and CTT. Our tool was

implemented in Python using the incremental SAT solver glucose3 [1]. Readers interested in reproducing the results of this section (Table 4, Table 5 and all related conclusions) can access the following Github repository: <https://github.com/PierreMartou/VaMoS2024-ReplicationPackage>.

7.1 Improvements analysis

To analyse the impact of each improvement on performance and on each other, Table 4 presents performance metrics for all possible combinations of improvements, where w_{look_ahead} is set to 0.5 and $w_{comparative}$ to 0.3. Each result is an average over 1000 generated test suites. These metrics include the **Size** of a test suite as well as the reconfiguration **Cost** (i.e. the number of (de)activations throughout a test suite, which is related to the test effort). Both metrics should be as low as possible. Standard deviation is added between parentheses, as it is an indication of stability.

Table 4: Size and cost with different improvements (standard deviation in parentheses).

Improvements	Size	Cost
None	18.4 (± 5.4)	70.9 (± 15.9)
Improvement 1	17.3 (± 5.2)	68.6 (± 15.7)
Improvement 2	14.4 (± 0.8)	58.6 (± 3.2)
Improvement 3	18.2 (± 5.1)	70.6 (± 15.7)
Improvements 1 & 2	12.6 (± 0.9)	53.8 (± 1.7)
Improvements 2 & 3	14.3 (± 0.8)	58.0 (± 3.3)
Improvements 1 & 3	17.3 (± 4.9)	68.5 (± 15.2)
Improvements 1 & 2 & 3	12.3 (± 0.6)	53.5 (± 1.5)

Only filtering out invertible interactions (Improvement 1) appears to have low impact on performance compared to the baseline algorithm, while the look-ahead mechanism (Improvement 2) is clearly the best-performing stand-alone improvement. However, in practice, the impact of the look-ahead heuristic can be shadowed by bias towards interaction coverage. This explains why filtering out invertible interactions (which reduces the weight of interaction coverage when computing scores) is most pertinent when combined with the look-ahead mechanism (Improvements 1 & 2).

Similarly, the comparative weight heuristic (Improvement 3) appears to have no impact when used alone. Nonetheless, the best-performing combination is still using all improvements together, both on size average and standard deviation. The comparative heuristic has lower impact on size and is more suited for performance stabilisation (as shown by the reduced standard deviation).

As for the cost, it varies proportionally to the size. A possible outcome was that even if the size of generated test suites was reduced, the overall number of transitions would remain the same (as if the test suite was made more “compact”) and hence the cost would remain high. However, the results show us that our generation algorithm does not only generate smaller test suites but also finds optimised transitions between configurations which reduces overall cost.

Together, these improvements reduce by 33% the size and by 25% the cost of generated test suites compared to the initial implementation of our combinatorial transition testing algorithm. They

also manage to drastically stabilise the generation algorithm, by reducing the standard deviation by 89%.

7.2 Validation with S.P.L.O.T.

Until now, we used our risk information system to analyse the shortcomings of CIT, defined combinatorial transition testing, and built an optimised generation algorithm. We now validate our testing approach on various and larger models, using the public feature model repository S.P.L.O.T. [20], more precisely their “Repository of Real Feature Models”. This repository contains freely distributed feature models, directly deposited by their authors or extracted from relevant sources such as academic publications. All of them are consistent (i.e. they have at least one valid configuration) with no dead features. They include cross-tree constraints of various impact. The cross-tree constraint ratio, or CTRC, is the ratio of the number of features in the cross-tree constraints to the number of features in the feature tree [21]. This ratio ranges from 0% to 100% across the different feature models. By using this repository, our goal is to test the resilience and performance of our generation algorithm against the complexity and diversity of the collected feature models. Note that it is unknown whether these feature models represent dynamically adaptive systems or not.

Table 5 presents our results using S.P.L.O.T. For clarity, we aggregated these feature models according to their number of features and indicated the categories in **#features**. We randomly sampled models among those available in the repository (at least one for each size if available). Less models are available for higher numbers of features, which explains the decreasing quantity in the categories (final quantity of models is reported under **Qty**). For each feature model within a category and for each different generation approach, 5 test suites are generated.

Our goal is to compare interaction testing (CIT) to transition testing, as well as to analyse the impact of our improvements, notably to assess scalability of our generation approach with regards to CIT (which was proven to have logarithmic growth [4]). **CTT 0** means using the generation algorithm of Section 6 with no improvement; **CTT 1** means using the first improvement; **CTT 1&2** the two first improvements; and **CTT 1&2&3** all improvements.

We base our comparison on the **size** and **cost** of generated test suites. We also compute the transition coverage when using CIT (**T.cov.**), i.e. the percentage of transitions covered, just like we did in Section 4. Note that as the reordering of a test suite affects its cost and transition coverage, we differentiate test suites reordered by maximising **dissimilarity** between test configurations (**CIT Diss**) and by minimising the reconfiguration **cost** (**CIT Cost**). Transition coverage for CTT is always 100% (as our generation algorithm does not end until all transitions are covered) and hence is not included.

First, we can observe that the impact of our improvements are very similar to those on our running example. Our conclusions are the same: the look-ahead heuristic (Improvement 2) has the largest impact on performance, and is further stabilised by the invertible interaction filter and comparative weight heuristic. Using all improvements (**CTT 1&2&3**) consistently yields the best results for transition testing by reducing the size by 29% on average, with the comparative heuristic having the lowest impact. Similarly, the cost and size of CTT-generated test suites grow proportionally and

Table 5: Analysis of the size, cost and transition coverage of test suites generated with combinatorial interaction testing (with different reordering) and transition testing (with different improvement combinations). All metrics are averaged over 5 generated test suites.

#features	Qty	Size					Cost						T. cov. (%)	
		CIT	CTT 0	CTT 1	CTT 1&2	CTT 1&2&3	CIT Diss	CIT Cost	CTT0	CTT 1	CTT 1&2	CTT 1&2&3	CIT Diss	CIT Cost
10-19	134	8.0	20.5	18.4	14.7	14.6	45.5	24.4	124.3	118.2	95.1	96.7	68.3	22.8
20-29	51	9.2	24.9	22.0	18.1	18.1	87.6	51.2	256.5	245.1	192.7	194.2	69.7	30.7
30-39	34	10.7	32.2	28.8	22.5	22.3	142.2	79.5	444.5	420.1	315.9	317.5	74.9	31.3
40-49	38	12.7	41.1	37.2	28.4	28.3	238.7	138.8	806.8	770.0	554.1	557.2	77.6	36.1
50-69	27	14.5	47.0	42.1	36.4	35.7	371.8	222.1	1225.3	1184.6	874.6	896.3	80.2	40.4
70-99	16	15.5	53.9	47.3	46.9	43.7	575.4	374.1	1798.8	1986.4	1847.6	1529.9	81.1	46.9

the improvements reduce the cost by 27% on average. The only improvement that does not always decrease the cost is the third improvement, as we observe that **CTT 1&2&3** has comparable or marginally higher costs compared to **CTT 1&2** (~ 1% higher).

As CTT-generated test suites have broader coverage, it was expected that they would have bigger size and cost than CIT-generated test suites. This is confirmed by the results. However, the most important criterion for feasibility is the scalability of our testing approach: if the size and cost of CTT-generated test suites are linearly correlated to CIT-generated ones, then combinatorial transition testing would have the same scalability as combinatorial interaction testing. Observing the values, they indeed seem to be linearly correlated, with an average factor of 2.08 between the sizes of test suites generated by CIT and CTT 1&2&3.

To further study their relation, we computed the Pearson correlation coefficient between the size of CIT-generated and CTT-generated test suites, according to the number of features in the feature models. When computing correlation, to avoid bias because of the higher number of feature models at lower numbers of features, results for models with the same number of features are averaged. This correlation coefficient is of ~0.987, where 0 means no correlation and 1 means total correlation. This experiment confirms that both are extremely closely correlated. We can conclude that combinatorial transition testing also grows logarithmically in the number of features while achieving transition coverage. Our answer to RQ3 is that the cost of transition coverage is the same as the cost of interaction coverage; if interaction coverage for a system is attainable, transition coverage should also be attainable.

Transition coverage for CIT-generated test suites reordered by dissimilarity does not exceed 80%, but is steadily increasing with the number of features. Even though we expect it to reach a maximum at some point, further studies are needed to validate the usefulness of transition testing on very large feature models (hundreds of features). The cost reordering (**CIT Cost**) which is most optimal for test effort reduction results in very low transition coverage.

Note that reaching even 80% coverage is not enough in testing. Missing faulty behaviour is unacceptable for critical systems whose safety cannot be compromised. In Section 4, we mentioned the possibility of combining multiple CIT-generated suites to reach 100% transition coverage. However, even at 80% coverage for each test suite, the overlapping coverage between test suites pushes the number of test suites to combine to at least ten (as was illustrated on

our case study in Table 3). The final cost and size of this approach would need to be multiplied by at least ten compared to standard CIT. Compare this to our transition testing approach which is only double the size and cost compared to standard CIT.

As a final validation, we observed the standard deviation of test suite sizes when generating multiple test suites for a same feature model. The results show that our improvements reduce this standard deviation by 81% on average on all models from S.P.L.O.T. (from **CTT 0** to **CTT 1&2&3**) which means that the improvements make results more stable.

8 THREATS TO VALIDITY

The focus of this paper was illustrating transition errors and providing a feasible solution for covering them on feature models with less than 100 features. Our proof of logarithmic growth for CTT was based on experimental comparison to CIT through hundreds of feature models. While this demonstrates its feasibility in practice in most cases, it is still not a mathematical proof (as is the case for CIT [4]). Bias remains a potential concern, and there is a theoretical possibility that specific feature models or a threshold on the number of features could affect the scalability of our testing approach.

We used S.P.L.O.T. for its extensive repository of realistic yet diverse feature models to challenge our generation algorithm. However whether they actually represent dynamically adaptive software systems is unknown. While a validation on only such systems would be more pertinent for our approach, a large enough repository of such feature models is currently unavailable to the best of our knowledge. While we do believe these feature models are diverse and representative enough, it could create a bias in our study.

Higher order transitions (i.e. with more than two features per transition) were not considered in our case study and validation. As for scalability, we do not have proof that higher order transition testing will grow logarithmically in terms of test effort. Nonetheless, we have good reason to believe it should work: all equations in Section 5 hold true for higher order transitions (i.e. the number of n -way transitions to cover is always less than or equal to the number of n -way interactions). We can expect linear correlations for higher values of n , maybe with a higher constant factor between n -CIT and n -CTT.

If the scope were to be expanded to very large feature models (hundreds of features), computation time scalability would also have to be studied for further validation. Specifically, the quantity and efficiency of SAT calls frequently act as significant performance bottlenecks. Although there is a correlation in terms of test effort between CIT and CTT, it is important to note that there is no evidence proving a similar correlation in computational aspects.

9 CONCLUSION AND FUTURE WORK

In this paper, we showed that combinatorial interaction testing struggles with complete transition coverage. Even the best test suite reordering does not achieve over ~80% transition coverage, and we would need to combine more than ten test suites together to find all potential behavioural transition errors (RQ1).

To fill this gap, we introduced combinatorial transition testing and devised a test generation algorithm which combines both interaction and transition testing. We proposed three improvements to an initial baseline algorithm which reduce both size (approx. -29%) and cost (approx. -27%) of generated test suites (RQ2). Finally, we validated its logarithmic growth in the number of features. It achieves full transition coverage with much less resources than what using CIT would need (RQ3). Combinatorial transition testing is proving to be a suitable complement to CIT to achieve a more comprehensive testing strategy for dynamically adaptive systems.

Whereas our running example was a dynamically adaptive system based on feature modelling, combinatorial transition testing could be generalised to other types of dynamically adaptive systems with potentially faulty transitions as well. As long as the system under test uses a modelling approach that is equivalent to a set of logical constraints, our testing approach can be used. For instance, studies have demonstrated how to reduce context-feature models [8, 9] to such a set [19]. These models incorporate contextual information to dynamically configure the behaviour of the system, a common characteristic in dynamically adaptive systems.

As changing the order of test scenarios modifies the transitions covered by them, transition testing inherently prevents reordering generated test suites. Reordering of test configurations has long been seen as an important step to optimise applications of test suite to real cases, with objectives such as reducing simulation efforts [19], early fault detection [25], or prioritisation of frequent behaviour [7]. To address this issue, prioritisation ought to be directly integrated into the generation process, rather than a posteriori. Promising directions include new heuristics in a greedy approach, weighted sampling approaches [2] or multi-objective algorithms (e.g. genetic algorithms [12]), to name a few.

Different steps of our generation algorithm could be improved in terms of computational efforts. For example, algorithms such as those presented by Johansen [13] and Kowal [15] could help identify dead/core features or other anomalies and alleviate future computations or SAT calls. A propagation step, which completes a partial configuration with values that are inferred by modern SAT solving, could speed up building new candidate test configurations in our test generation approach [19].

REFERENCES

- [1] Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. 2013. Improving glucose for incremental SAT solving with assumptions: Application to MUS

- extraction. In *Theory and Applications of Satisfiability Testing–SAT 2013: 16th International Conference, Helsinki, Finland, July 8–12, 2013. Proceedings 16*. Springer, 309–317.
- [2] Eduard Baranov, Axel Legay, and Kuldeep S Meel. 2020. Baital: an adaptive weighted sampling approach for improved t-wise coverage. In *ESEC/FSE 2020*. 1114–1126.
- [3] Gagatay Catal and Deepti Mishra. 2013. Test case prioritization: a systematic mapping study. *Software Quality Journal* 21, 3 (2013), 445–478.
- [4] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. 1997. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering* 23, 7 (1997), 437–444.
- [5] Myra B Cohen, Matthew B Dwyer, and Jiangfan Shi. 2008. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Transactions on Software Engineering* 34, 5 (2008), 633–650.
- [6] Ismaile de Sousa Santos, Rossana Maria de Castro Andrade, Lincoln Souza Rocha, Santiago Matalonga, Kathia Marcal de Oliveira, and Guilherme Horta Travassos. 2017. Test case design for context-aware applications: Are we there yet? *Information and Software Technology* 88 (2017), 1–16.
- [7] Xavier Devroey, Gilles Perrouin, Maxime Cordy, Hamza Samih, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. 2017. Statistical prioritization for software product line testing: an experience report. *Software and Systems Modelling* 16, 1 (2017), 153–171.
- [8] Benoît Duhoux. 2022. *Feature-Based Context-Oriented Software Development*. Ph.D. Dissertation.
- [9] Benoît Duhoux, Kim Mens, Bruno Dumas, and Hoo Sing Leung. 2019. A Context and Feature Visualisation Tool for a Feature-Based Context-Oriented Programming Language. In *SATTOSE '19 (CEUR Workshop Proceedings, Vol. 2510)*. CEUR-WS.org.
- [10] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding flaky tests: The developer's perspective. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 830–840.
- [11] Achiya Elyasaf, Eitan Farchi, Oded Margalit, Gera Weiss, and Yeshayahu Weiss. 2022. Combinatorial sequence testing using behavioral programming and generalized coverage criteria. *arXiv preprint arXiv:2201.00522* (2022).
- [12] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. 2013. Multi-objective test generation for software product lines. In *SPLC '13*. 62–71.
- [13] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2012. An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models. In *Proceedings of the 16th International Software Product Line Conference - Volume 1 (SPLC '12)*. ACM, 46–55.
- [14] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report. Carnegie-Mellon University Software Engineering Institute.
- [15] Matthias Kowal, Sofia Ananieva, and Thomas Thüm. 2016. Explaining Anomalies in Feature Models. *ACM SIGPLAN Notices* 52, 3 (2016), 132–143.
- [16] D Richard Kuhn, James M Higdon, James F Lawrence, Raghu N Kacker, and Yu Lei. 2012. Combinatorial methods for event sequence testing. In *ICST '12*. IEEE, 601–609.
- [17] D Richard Kuhn and Michael J Reilly. 2002. An investigation of the applicability of design of experiments to software testing. In *27th Annual NASA Goddard*. IEEE, 91–95.
- [18] Pierre Martou, Benoît Duhoux, Kim Mens, and Axel Legay. 2023. Beyond Combinatorial Interaction Testing: On the need for transition testing in dynamically adaptive context-aware systems. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 100–104.
- [19] Pierre Martou, Kim Mens, Benoît Duhoux, and Axel Legay. 2022. Test scenario generation for feature-based context-oriented software systems. *JSS* (2022), 111570.
- [20] Marcilio Mendonca, Moises Branco, and Donald Cowan. 2009. S.P.L.O.T.: Software Product Lines Online Tools. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, 761–762.
- [21] Marcilio Mendonca, Andrzej Wąsowski, and Krzysztof Czarnecki. 2009. SAT-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference*. 231–240.
- [22] Changhai Nie and Hareton Leung. 2011. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)* 43, 2 (2011), 1–29.
- [23] Ana B Sánchez, Sergio Segura, and Antonio Ruiz-Cortés. 2014. A comparison of test case prioritization criteria for software product lines. In *ICST '14*. IEEE, 41–50.
- [24] Ronny Seiger and Thomas Schlegel. 2012. Test modeling for context-aware ubiquitous applications with feature petri nets. In *MODIQUITOUS*. Citeseer.
- [25] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *STVR* 22, 2 (2012), 67–120.