# Combinatorial Transition Testing in Dynamically Adaptive Systems: Implementation and Test Oracle

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

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

Kim Mens
ICTEAM, UCLouvain
Louvain-la-Neuve, Belgium
kim.mens@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, combinatorial interaction testing (CIT) can create concise test suites covering all valid pairs of features of such systems. While CIT claims to find all errors caused by two features, it does not cover certain errors occurring only for specific *transitions* between features. To address this issue we study the technique of Combinatorial Transition Testing (CTT), which includes both generation and detection of what we call behavioural transition errors. From an initial generation algorithm that combines both interaction and transition coverage but lacks scalability, we propose an optimised version that enables CTT even for hundreds of features. From a valid test suite covering all transitions, we complete our testing approach with a test oracle that detects all behavioural transition errors without any prior knowledge of the system's behaviour. After a comprehensive analysis over a large number of feature models, we conclude that size of CTT-generated test suites and test effort needed to use our test oracle are linearly correlated to CIT-generated ones and that CTT 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; test oracle; error detection; dynamically adaptive software systems; dynamic software product lines; feature modelling; software testing; behavioural transition errors


*JSS 2024, ,*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

**ACM Reference Format:**
Pierre Martou, Benoît Duhoux, Kim Mens, and Axel Legay. 2024. Combinatorial Transition Testing in Dynamically Adaptive Systems: Implementation and Test Oracle. In *Proceedings of Journal of Systems and Software (JSS 2024).* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 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 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 thus 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 [18, 25]. Furthermore, recent research has applied CIT to feature-based context-oriented systems [21], a particular class of dynamically adaptive software systems [9].

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 detect this error or fault. We 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). Such a perfect test oracle would need to know the expected behaviour for every possible test configuration. However, listing every possible behaviour of

dynamically adaptive software systems suffers from the same problem as testing every valid configuration: it is infeasible in practice due to combinatorial explosion.

This paper will tackle the following five research questions:

***RQ1**: Which behavioural* transition *errors can CIT detect?* In an earlier workshop paper [19] we 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 the simultaneous activation or deactivation (called transitions) of a set of features at runtime. 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, 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 address RQ1 in more depth and argue that CIT cannot easily detect all behavioural transition errors.

Our analysis shows that although combining enough test suites generated by CIT could achieve full transition coverage, the cost can be enormous and highly variable. Let us take an example with two features $A$ and $B$ whose value is either 0 or 1, with no additional constraints. A CIT-generated test suite will contain the following 4 configurations in any order, where each configuration gives a value to {A, B}: {0, 0}, {0, 1}, {1, 0}, {1, 1}. To capture all possible behavioural transition errors we would like to achieve full transition coverage, i.e. each possible pair of feature switches (activating both features $A$ and $B$, deactivating both $A$ and $B$, activating $A$ while deactivating $B$, and deactivating $A$ while activating $B$) must appear at least once in the test suite. To achieve that, we determined experimentally that, on average, we need to combine 7.5 randomly-ordered test suites that contain these 4 configurations, or a total size of 30 test configurations. However, an optimised test suite of size only 6 can be found for the same result: {0, 0}, {1, 1}, {0, 0}, {0, 1}, {1, 0}, {0, 1}. This is a reduction of 80% in total size, and finding such an optimised test suite is the goal of Combinatorial Transition Testing.

***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 first propose an initial (greedy) generation algorithm that validates the pertinence and feasibility of CTT. Then we compare it with an improved version that includes three optimisations to enhance the algorithm's efficiency.

We need to emphasise that, while detecting such behavioural transition errors is important for dynamically adaptive systems, whose configurations can change dynamically by (de)activating features at runtime, it is not relevant for non-dynamically adaptive systems, where behavioural transitions typically do not occur.

***RQ3**: What is the test effort of adding transition coverage to interaction coverage?* We conducted a quantitative study to compare the coverage and test efforts between CIT and CTT. For this we used the public feature model repository S.P.L.O.T. [23]. Based on this study we can 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 demonstrates that our optimisations not only enhance efficiency but also enable Combinatorial Transition Testing. They significantly reduce the average size and cost of generated test suites, as compared to running without optimisations. Moreover, the original algorithm can hit a dynamic time-out (set to five times the size of test suites generated by CIT for each feature model) for many of the analysed feature models, ranging from a quarter of small feature models to nearly all models when approaching 100 features. In these scenarios, the initial algorithm keeps producing new test configurations while failing to achieve full transition coverage until it hits the time-out limit. Our optimisations effectively eliminate this issue.

***RQ4**: How can we detect possible behavioural transition errors during execution?* We present a test oracle that enables comprehensive detection of behavioural transition errors without prior knowledge of the system's behaviour. Using a test suite with full transition coverage, we create alternative execution paths between each consecutive test configuration that omit specific transitions capable of causing errors. If the software system exhibits any behaviour or internal state different from the original execution path when using the alternative paths, the test oracle can affirm that a transition may be responsible for the unexpected behaviour.

***RQ5**: What is the test effort of using a test oracle to detect possible behavioural transition errors?* Just like for RQ3, we use the public feature model repository S.P.L.O.T. [23] to validate the feasibility of our test oracle. We experimentally proved that the test effort needed to use our test oracle is linearly correlated to CTT (size and number of alternative execution paths) and also grows logarithmically in the number of features. Whereas it is the case that for some transitions, called undecomposable, our test oracle cannot detect whether they could trigger a behavioural transition error, we do show that such transitions are rather uncommon (less than 4% of all transitions) and quickly become irrelevant as the number of features increases (0.4% when approaching 100 features).

*Novelty.* This paper extends our earlier paper [20] by completing our Combinatorial Transition Testing approach. A test suite with full transition coverage is ensured to trigger all behavioural transition errors of a software system. In practice, however, we also need to identify *which* behavioural transition are erroneous. This paper fills that gap by proposing a test oracle compatible with our generated test suites (new research question RQ4). We also validated the performance of our test oracle against hundreds of realistic feature models from the public feature model repository S.P.L.O.T. (new research question RQ5).

In addition to the novel contributions tackled by those two new research questions (RQ4 and RQ5), we also largely improved our exploration of research question RQ2. We significantly enhanced the algorithm's performance by adapting the look-ahead mechanism (cf. Subsection 6.3) to be more "realistic". The improved look-ahead mechanism will now correctly favour a large future transition coverage over a small transition coverage, whereas it would previously exaggerate its predictions with unrealistic expectations for future transition coverage. We also further increased the readability of

our test suite generation algorithm by illustrating it with detailed pseudo code (cf. Subsection 6.5).

This led us to modify and expand upon our answer to research question RQ3 as well. We analyse deeper the impact of the weights used in our different heuristics with additional graphs (cf. Subsection 8.1) and perform better hyper-parameter tuning. Last but not least, we extended our analysis of the test effort of our approach (RQ3) with more complex feature models (i.e. with more *alternative* constraints in the feature models), which was not the case in our previous paper. This extension brings greater diversity to the feature models. Our analysis shows that with those more complex models the initial naive algorithm often hits our dynamic time-out limit. This issue impacts a quarter of the small feature models and affects all models approaching 100 features. Our optimised algorithm no longer suffers from this issue, however. This new result highlights the importance of the impact that our optimisations have.

*Structure.* The remainder of 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 easily find *behavioural transition errors* in such systems. Section 5 introduces the main concepts of our *Combinatorial Transition Testing* (CTT) approach. Section 6 then presents our optimised greedy algorithm. Section 7 illustrates and proposes a test oracle to detect unexpected behavioural errors. Section 8 analyses the performance of these optimisations, compares CTT to standard CIT and evaluates the cost of using our test oracle. Section 9 discusses threats to the validity of our findings. Section 10 concludes and discusses remaining challenges to fully turn CTT into a comprehensive testing technique.

## 2 BACKGROUND AND RELATED WORK

### 2.1 Combinatorial Interaction Testing

Let us first describe the vocabulary used in this paper, inspired by Combinatorial Interaction Testing and feature modelling. Given a feature $f$, a function $v : f \rightarrow \{activated, deactivated\}$ assigns a specific value $v_f$ to a feature $f$. A partial test configuration is defined as a set of value assignments $\{v_1, ..., v_n\}$ for a set of features $\{f_1, ..., f_n\}$ 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 {-High, -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) $\{v_1, ..., v_n\}$ 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 {+High, -Low, +Map} covers the interactions {+High, -Low} and {+High, +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.

### 2.2 Transition testing

Errors related to the evolution of configuration parameters are known issues in developing software [28]. Such errors can impact continuous support, continuous integration or continuous adaptation. Ideally, a software system should behave predictably, in the sense that the impact of setting a specific configuration on a software system should be the same regardless the system's previous state.

An alternative approach to cover all transitions is that of State Transition Testing [7], a technique which is typically used on systems that explicitly represent transitions between states (e.g., using Petri Nets [27]). 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 [12, 17] 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 (≃ events) they can generate. For a single configuration, however, this approach could be used to verify different activation orders of features (≃ 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 [15].

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

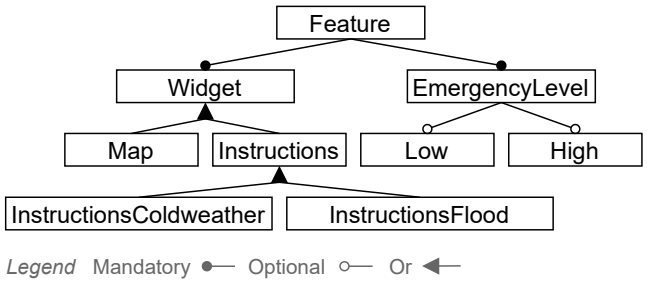


**Figure 1: Feature model of a simplified version of the Risk Information System.**

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 *Emergency Level* 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 there is 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 [11] 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 and not through any other sequence of transitions. This idea forms the basis of our test oracle in Section 7.

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 [5, 6]. 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 [6, 21]. 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 [30]. 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 [4, 26]. 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 [21].

**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 [21] (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 8, 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

transitions[1]. 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 ~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 easily 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 $|v_1, ..., v_n|$ if, in two consecutive configurations, the second configuration includes the assignment of values $\{v_1, ..., v_n\}$ while the first configuration includes the opposite assignment, i.e., $\{-v_1, ..., -v_n\}$.*

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:

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

Definition 2. *We say that a transition $|v_1, ..., v_n|$ is valid if the specified value assignment $\{v_1, ..., v_n\}$ and its opposite $\{-v_1, ..., -v_n\}$ 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 $|v_1, ..., v_n|$ implies covering the corresponding interaction $\{v_1, ..., v_n\}$. 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.

$$\#invert\_int_n = \#transitions_n \tag{1}$$

$$\#invert\_int_n + \#non\_inv\_int_n = \#interactions_n \tag{2}$$

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

$$\#transitions_n + \#non\_inv\_int_n = \#interactions_n \tag{3}$$

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, as illustrated before, 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 behavioural interaction and transition errors more efficiently.

The foundation of the algorithm is a row-by-row greedy algorithm for highly-constrained systems [6]. 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 $f_2, \ldots, f_k$. 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 $f_1, \ldots, f_{n-1}$ have been assigned to $v_1, \ldots, v_{n-1}$. 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 10.

## 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%.

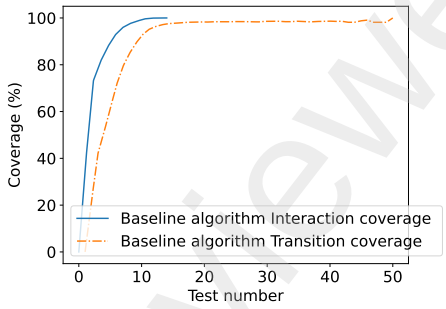


**Figure 2: Evolution of the interaction/transition coverage of a test suite (averaged over 1000 test suites)**

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 configuration, 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 configuration 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 not 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 $f$, $\#(uncoverable\_t_{f,v})$ 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 $s\_la_{f,v}$ is:

$$s\_la_{f,v} = \#(uncovered_{f,v}) + w_{look\_ahead} * \#(uncoverable\_t_{f,-v}) \quad (4)$$

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.

However, when the number of features increases, this mechanism tends to promote future transitions rather than continuing to progress through transition coverage. In other words, the generation algorithm will only produce test configurations whose goal is to prepare for the future without ever actually achieving it. The reason for this is as follows: while these test configurations do make it possible to cover a large number of potential future transitions, these future transitions are not necessarily compatible with each other. Even if a test configuration can cover 100 different transitions, it is possible that only 20 of them can realistically be covered in the next test configuration.

To prevent this, we propose a more realistic look-ahead mechanism that ensures compatibility between future transitions. When computing the score of a value, a valid partial configuration (checked through SAT solving) is built and is used to verify that all prepared transitions are compatible. Whenever this value would prepare for a future transition, a weighted score is added only if the values in this transition can be integrated into the partial configuration while keeping it valid.

The final pseudo code of our realistic look-ahead mechanism is shown in Subsection 6.5.

## 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 $s\_la_{f,v}$ 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 :

$$s\_comp_{f,v} = s\_la_{f,v} - w_{comparative} * s\_la_{f,-v} \quad (5)$$

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. Pseudo-code will be shown in Subsection 6.6 to illustrate the final score computations.

## 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 configuration. 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 $f_2, \ldots, f_k$. 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 $f_1, \ldots, f_{n-1}$ have been assigned to $v_1, \ldots, v_{n-1}$. 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.

## 6.6 Pseudo-code of optimised algorithm

Now that we have provided our optimised algorithm, we will illustrate it with pseudo code. Listing 3 shows the complete overview of our algorithm. Lines 2-10 pre-process valid interactions and transitions to cover with the non-invertible interaction filter (Improvement 1). Lines 12-32 are the main body of our algorithm to create the test suite, initialised in Line 12. Until the two lists of uncovered transitions and interactions are empty, our algorithm computes the best candidate configuration at each iteration and adds it to the test suite currently being constructed. Lines 14-27 correspond to the generation of candidate configurations based on our three steps, as described above. Lines 28-32 select the best candidate to add to the test suite based on the score computation and update the transitions and interactions left to cover. Note that the previous configuration, initialised in Line 13, is important to build the current test configuration. Without this contextualised information, we cannot cover transitions (Definition 1).

```
1  def generate_test_suite(feature_model):
2    # Preprocessing with non-invertible filter
3    uncovered_transitions = []
4    uncovered_interactions = []
5    for each pair of values in feature_model:
6      if pair is valid:
7        if opposite(pair) is valid:
8          add pair to uncovered_transitions
9        else:
10         add pair to uncovered_interactions
11   # Main body of algorithm starts here
12   test_suite = []
13   prev_config = EmptyConfiguration()
14   # generation of candidate configurations
15   while uncovered_transitions not empty or
     uncovered_interactions not empty:
16     candidates = []
17     for count=1 to nb_candidates:
18       new_candidate = EmptyConfiguration()
19       # Step 1
20       best_value = compute_first_value(
         prev_config)
21       add best_value to new_candidate
22       # Step 2
23       for feature in shuffle(feature_model\{
         first feature}):
24         # Step 3
25         best_value = compute_best_value(feature,
           new_candidate, prev_config)
26         add best_value to new_candidate
27         add new_candidate to candidates
28     # select the best candidate
29     best_candidate = compute_best_candidate(
       candidates, prev_config)
30     add best_candidate to test_suite
31     prev_config = best_candidate
32     update uncovered_transitions,
       uncovered_interactions
33   # return generated test suite
34   return test_suite
```

**Listing 3: Pseudo-code of complete optimised algorithm**

In Listing 3, Line 20 computes the first value to be selected in a new candidate configuration based on the previous test configuration. This corresponds to Step 1 of our algorithm and is illustrated by the pseudo code in Listing 4. Lines 2-5 in Listing 4 calculate a score for each possible value (i.e., activated and deactivated) for each feature in the feature model. Line 6 then returns the best (feature, value) pair with the highest score. Ties are broken at random.

The score for each value is calculated according to the possibility of generating a test configuration with that value. Lines 10-11 increment the score by this value for each uncovered interaction that includes this value. Lines 14-16 increment its score for each uncovered transition that contains this value and for which the previous test configuration contains this opposite transition (Definition 1). Finally, lines 17-19 can also add a look-head weight to the score of this value only if the algorithm can prepare a future transition based on this value to cover uncovered transitions in future (i.e., in the next test configuration). To that end, a partial configuration *valid_future* is initialised on line 13. This partial configuration allows to simulate a potential "future" test configuration that remains valid by satisfying all the constraints of the feature model. This refers to the realistic look-ahead mechanism of Subsection 6.3.

```
1  def compute_first_value(prev_config):
2    scores = []
3    for feature in feature_model:
4      add compute_first_score(feature_activated,
       prev_config) to scores
5      add compute_first_score(feature_deactivated,
        prev_config) to scores
6    return best value according to scores
7
8  def compute_first_score(value, prev_config):
9    score = 0
10   for interaction in uncovered_interactions that
     includes value:
11     increment score
12
13   #realistic look-ahead mechanism
14   valid_future = new Configuration()
15   for transition in uncovered_transitions that
     includes value:
16     if opposite(values(transition)) in
       prev_config:
17       increment score
```

```
18      if value in opposite(values(transition)) and
         is_valid(valid_future ∪ values(transition)):
19        valid_future = valid_future ∪ values(transition
          )
20        add look-ahead weight to score
21    return score
```

**Listing 4: Computation of the first value in the generation of a new candidate**

Analysing the pseudo code of Listing 3 further, Line 25 determines the best value for a feature (i.e., if a feature must be activated or deactivated in the current configuration) when building a candidate configuration. This computation refers to Step 3 of our complete algorithm and is detailed by the pseudo code in Listing 5. In this pseudo code, lines 1–9 compute the best value for a feature. Lines 2–5 filter invalid configurations. It is not necessary to calculate a score to identify the best value for a characteristic if a specific value for this feature leads to an invalid configuration. In this case, the opposite value (leading to a valid configuration) is returned as the best value. If both values are possible, lines 7–9 compute the score for each value, taking into account the current candidate test configuration and the previous test configuration, and return the value with the higher score. In case of a tie, it is broken randomly.

The score to identify which value is the best for a feature is calculated as follows. Lines 11–15 compute the score associated to interactions that would be covered by adding this value to the current candidate. Similarly, lines 18–21 increment score according to the number of transitions that would be covered by using this value. Lines 23–25 then add score according to the look-ahead mechanism, while building a *valid_future* configuration whose goal is to keep the expectations realistic.

```
1  def compute_best_value(feature, curr_config,
   prev_config):
2    if curr_config ∪ feature_activated is invalid:
3      return feature_deactivated
4    if curr_config ∪ feature_deactivated is invalid:
5      return feature_activated
6
7    score_activated = compute_score(feature_activated,
     curr_config, prev_config)
8    score_deactivated = compute_score(feature_deactivated
     , curr_config, prev_config)
9    return value associated to best score
10
11 def compute_score(value, curr_candidate,
   prev_config):
12   score = 0
13   for interaction in uncovered_interactions that
     includes value:
14     if interaction is covered by curr_candidate ∪
       value:
15       increment score
16
17   #realistic look-ahead mechanism
18   valid_future = new Configuration()
19   for transition in uncovered_transitions:
20     if transition is covered by the sequence
       prev_config then curr_candidate ∪ value:
21       increment score
22
23     if opposite(values(transition)) in
       curr_config ∪ value and is_valid(valid_future
       ∪ values(transition)):
24       valid_future = valid_future∪values(transition)
25       add look-ahead weight to score
26
27   return score
```

**Listing 5: Computation of a score with look-ahead and comparative mechanisms**

Finally, when all candidate test configurations have been generated, the algorithm must choose the best of them based on the previous test configuration, as shown on line 29 in Listing 3. This decision is illustrated in the pseudo-code of listing 6. Lines 5–7 and Lines 10–12 increment the score of the candidate according to the number of interactions and transitions it would cover, respectively. Lines 15–17 increase the score of the candidate depending on the number of transitions it would cover based on the realistic look-ahead mechanism. Line 20 returns the best candidate for the test configuration being built. All ties are broken at random. Note that the first generated test configuration omits all score computations related to transitions as there is no previous configuration.

```
1  def compute_best_candidate(candidates,
   prev_config):
2    scores = {}
3    for candidate in candidates:
4      score = 0
5      for interaction in uncovered_interactions:
6        if interaction is covered by candidate:
7          increment score
8
9      valid_future = new Configuration()
10     for transition in uncovered_transitions:
11       if transition is covered by the sequence
         prev_config then candidate:
12         increment score
13
14       # look-ahead mechanism
15       if is_valid(valid_future ∪ values(transition
         )) and transition is covered by the
         sequence candidate then valid_future ∪
         values(transition):
16         valid_future = valid_future ∪ values(
           transition)
17         add look-ahead weight to score
18
19     add (score, candidate) to scores
20   return best candidate according to score
```

**Listing 6: Best candidate computation algorithm**

## 7 TEST ORACLE FOR TRANSITION ERRORS

The previous section showed how to generate test suites with full transition coverage, whose execution will trigger all behavioural transition errors of a software system. Nonetheless we also need to identify which behavioural transitions are erroneous. In practice, just like it is infeasible to test every single valid configuration, it is also infeasible to define what is the correct behaviour for every single valid configuration. This brings us to RQ4: *How can we detect possible behavioural transition errors during execution?* To answer this research question, we create a test oracle. It takes as input a generated test suite with full transition coverage, and generates a set of alternative execution paths that will reveal any behavioural transition errors, once executed in parallel to the original test suite. The key insight is that for the system behaviour to be predictable, setting a specific configuration on a software system should lead to a similar observable behaviour regardless of the execution path leading to it.

In Subsection 7.1, we first define the concepts of our test oracle, before describing our generic algorithm for this oracle in Subsection 7.2.

### 7.1 Definitions of test oracle

We split the definition of our test oracle in two parts. We first define how we can detect an erroneous transition, and then describe how to generate alternative execution paths to find this error. To illustrate these different concepts, we revisit our case study of Section 3.

*7.1.1 Detecting erroneous transitions.* In a test suite with full transition coverage, all valid transitions will occur at least once during its execution. However, some transitions may lead to errors. A naive solution to detect erroneous transitions could be to list the correct behaviour for each valid configuration and compare the actual execution to these expected behaviours. Nevertheless, such an approach is infeasible due to a combinatorial explosion of possibilities. Therefore, we propose the usage of a test oracle to detect such errors.

For a transition generated by the test suite, our oracle aims to run the original execution path and an alternative execution path for that transition. It then compares the behaviour of both execution paths and when they differ the behavioural transition is considered erroneous.

Let us revisit our running example to illustrate such a test oracle. In Section 3.2, we illustrated the behavioural transition error caused by the transition |-High, +Low|. We showed that the emergency level state was erroneously set to 2 instead of 1 when performing this transition. In other words, with a start configuration $C_{start}$ that includes {+High, -Low}, performing this particular transition will result in a final configuration $C_{final}$ that includes {-High, +Low} but that contains an erroneous internal state, causing an incorrect system behaviour. We will call this the original execution path.

In this example, the objective of the test oracle is to detect that the emergency level is set to an unexpected value. To that end, we assume the emergency level will correctly be set to 1 when performing any alternative sequence of transitions that excludes the transition |-High, +Low| from a start configuration $C_{start}$ {+High, -Low} to a final configuration $C_{final}$ {-High, +Low}. Note that these alternative sequences of transitions may pass through additional intermediate states (intermediate configurations) to go from the same start configuration to the same final configuration as the original execution path.

Here, we can imagine that such an alternative execution path contains the sequence of two distinct transitions |+Low| and |-High|, with one intermediate configuration $C_1$. Performing the original and alternative execution paths would result in a different internal state and behaviour of the software system, and would thus allow us to detect that the transition leads to a possible error. This is illustrated in Figure 3, where the internal state is set to 2 and 1 for the original and the alternative execution paths, respectively. If there was no erroneous transition, there should have been no difference between the two internal states.

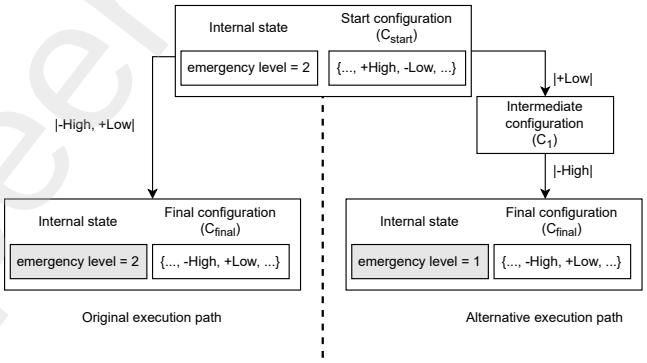


**Figure 3: Parallel execution paths of the test oracle leading to different system behaviour**

The concept of internal state is described in Definition 4. We can then define how a transition can be detected as erroneous or not by our test oracle in Definition 5.

Definition 4. *Features are program pieces that can be (de)activated. The internal state is the result of a particular system configuration in a dynamically adaptive software system, typically a set of observable variables and observable behaviours. Two internal states can be compared, and must be the same if they are associated to the same system configuration.*

Definition 5. *Given an original execution path that contains a transition under test, if we can define an alternative execution path that excludes this transition, a behavioural transition error can be detected by comparing the resulting internal state of both the original and alternative execution paths. If there is a difference, this transition is the cause of a behavioural transition error.*

This definition does not exclude the possibility of having multiple potentially erroneous transitions in a same original execution path, and excluding all of them in the corresponding alternative

execution path. If there is no observed difference in the system behaviour, all of these transitions are theoretically correct. If a difference would be observed, we could then split the potentially erroneous transitions in two sets of "original" execution paths and generate new alternative execution paths that respectively exclude the first then the second half of potentially erroneous transitions. We can then see which half results in a different system behaviour and repeat this process until one or multiple erroneous transitions are isolated.

Furthermore, the observed difference might also have a more complex cause. For example, it may be caused by different transitions in the alternative execution paths instead of in the excluded transition. However, once a difference has been detected, the role of the test oracle is complete: it has detected whether or not a potential error occurs during execution, which we cannot even know without a test oracle. Defining the correct behaviour for one end configuration and identify which of the original or alternative execution paths is faulty is an easy problem compared to the one of defining correct behaviours for every valid configuration.

*7.1.2 Alternative execution paths for all transitions.* Given Definition 5, Definition 6 defines what would be full error detection.

Definition 6. *A test oracle can detect all behavioural transition errors (i.e. achieve full error detection) in a dynamically adaptive software system if for all valid transitions in the system, there exists at least one original execution path that includes it and a corresponding alternative execution path that excludes it.*

A test suite with full transition coverage gives us exactly what we need to achieve full error detection: if we consider this test suite as the original execution path, then our goal is to generate alternative execution paths for each pair of consecutive test configurations such that all transitions are compared at least once between an original execution path that includes it and an alternative one that excludes it. All transitions will be included at least once in an original execution (this follows from our definition of a test suite with full transition coverage) and excluded at least once in the corresponding alternative one.

Table 4 illustrates our test oracle on our case study. The original execution path was generated by the approach described in Section 6. Generation of the alternative execution paths is discussed in the next section.

First, we can show that the erroneous transition |-High, +Low| is covered by the original execution path between test configurations 2 and 3. Its associated alternative execution path excludes this transition, and the resulting internal states will be different even if they transition towards the same configuration.

Let us emphasise that a transition needs only one comparison where an original execution path includes it while one associated alternative one excludes it. This is the reason why the transition |-High, +Low|, whose first occurrence is between test configurations 2 and 3, is excluded in the associated alternative execution path, but is not excluded in its second occurrence between test configurations 5 and 6. As a consequence, the alternative execution path between test configurations 5 and 6 can (and does) include it to gain in efficiency, i.e. shorter size of execution path and lower cost (defined as the (de)activation of features).

An observer of Table 4 will notice that the execution path between test configurations 3 and 4 has two alternative execution paths. The reason is that a single alternative execution path which excludes all transitions under test (i.e. those not yet detected by the test oracle) does not exist. More specifically, we cannot define a valid *alternative* execution path (that goes through only valid configurations) which excludes both the transitions |+Instructions, +InstructionsFloods| and |+Instructions, +InstructionsColdweather|, since activating the feature *Instructions* will always either activate its child feature *InstructionsFloods* or *InstructionsColdweather*, due to their *OR* relationship (Figure 1). To solve this issue, we need to split the transitions under test in two groups and define an alternative execution path for each group.

Non-satisfiability of alternative executions paths can go further. Some transitions can never be avoided between two different configurations. We formalise and identify those transitions in Definition 7.

Definition 7. *Given an original execution path where the transition $|v_1, ..., v_n|$ is under test, if an alternative execution path which excludes this transition does not exist, behavioural errors caused by this transition are said to be* undetectable *and this transition is said to be* undecomposable.

Finally, between test configurations 7 and 8, no alternative execution path is generated. The reason is that no new transition is covered between test configurations 7 and 8; test configuration 8 exists only to *prepare* for future transitions that are currently uncoverable. This is a result of our realistic look-ahead mechanism, which allowed the generation to find a test configuration (test configuration 8) from which it could efficiently cover the last remaining uncovered transitions.

## 7.2 Test oracle generalisation

We now generalise our test oracle to any dynamically adaptive software system that uses feature modelling. From a test suite with full transition coverage, our goal is to generate alternative execution paths for each pair of consecutive test configurations such that each transition is either detected once by our test oracle (Definition 6), or found to be undecomposable (Definition 7).

Consider a test suite $T$ of length $N$ with full transition coverage, where test configuration $i$ is noted $c_i$, an empty list which records the transitions already detected and another empty list to record the undecomposable transitions.

We generate $N - 1$ groups of alternative execution paths $P_i$ ($i$ ranging from 1 to $N - 1$). $P_i$ is associated to the pair of consecutive test configurations $[C_i, C_{i+1}]$.

Each group of alternative execution paths $P_i$ is generated along a *divide-and-conquer* scheme in order to generate valid alternative execution paths and isolate undecomposable transitions. The following steps detail our strategy:

(1) Compute the list of all transitions between $C_i$ and $C_{i+1}$. Remove all transitions already detected in previous iterations and all undecomposable transitions found until now. This list initialises the original set of transitions under test and the list of transitions currently under test $T_{undertest}$.
(2) Let $T_{undertest}$ be the list of transitions currently under test. Try generating an alternative execution path between $C_i$ and $C_{i+1}$ which excludes all transitions in $T_{undertest}$.

**Table 4: Full description of the original execution path (generated by CTT) and the corresponding alternative execution paths (generated by our test oracle) in terms of features (de)activations between test configurations for our case study. Elements with bigger, bold and italic font are those mentioned in the text.**

| N° | Original execution path | Alternative execution paths | |
|---|---|---|---|
| 0→1 | +Widget, +InstructionsFloods, +Low, +Map, +EmergencyLevel, +Instructions | *No transition coverage in the first test configuration.* | |
| 1→2 | -InstructionsFloods, -Low, -Map, +InstructionsColdweather, +High | -Map | |
| | | -Low | |
| | | +InstructionsColdweather | |
| | | -InstructionsFloods | |
| | | +High | |
| 2→3 | -InstructionsColdweather, -Instructions, ***-High, +Low***, +Map | -High | |
| | | +Map | |
| | | -InstructionsColdweather, +InstructionsFloods | |
| | | -InstructionsFloods, -Instructions | |
| | | +Low | |
| 3→4 | -Low, -Map, +InstructionsColdweather, +InstructionsFloods, +Instructions, +High | -Low, +High | +InstructionsFloods, +Instructions |
| | | +Instructions, +InstructionsColdweather | -Map, -Low, +High, +InstructionsColdweather |
| | | -Map, +InstructionsFloods | |
| 4→5 | -InstructionsColdweather, -InstructionsFloods, -Instructions, +Map | -InstructionsFloods, -High | |
| | | -Instructions, -InstructionsColdweather, +High, +Map | |
| 5→6 | -Map, ***-High, +Low***, +Instructions, +InstructionsColdweather, +InstructionsFloods | ***-High, +Low*** | |
| | | -Map, +InstructionsFloods, +Instructions, +InstructionsColdweather | |
| | | | |
| 6→7 | -InstructionsColdweather, -InstructionsFloods, -Low, -Instructions, +Map, +High | -InstructionsColdweather, -InstructionsFloods, -Instructions, +Map | |
| | | -Low, +High | |
| | | | |
| 7→8 | -Map, +InstructionsColdweather, +Low, +Instructions | ***No alternative execution path is necessary.*** | |
| 8→9 | -InstructionsColdweather, -Low, -High, +InstructionsFloods, +Map | -InstructionsColdweather, -Instructions, -High, +Map | |
| | | -Low, +InstructionsFloods, +Instructions | |
| 9→10 | -InstructionsFloods, +InstructionsColdweather, +Low, +High | +Low | |
| | | -InstructionsFloods, +High, +InstructionsColdweather | |
| 10→11 | -InstructionsColdweather, -Map, +InstructionsFloods | -InstructionsColdweather, -Low, -Instructions, -High | |
| | | -Map, +High, +InstructionsFloods, +Instructions, +Low | |
| 11→12 | -InstructionsFloods, -High, +InstructionsColdweather, +Map | -Low, -High, +Map | |
| | | -InstructionsFloods, +Low, +InstructionsColdweather | |

(3) If the previous step succeeded in generating an alternative execution path, add it to $P_i$. Add all transitions in $T_{undertest}$ to the list of detected transitions.
(4) If no alternative execution path exist and $T_{undertest}$ contains more than one transition, split the transitions $T_{undertest}$ in two distinct groups $T_{undertest,1}$ and $T_{undertest,2}$. Repeat from the step 2, once with $T_{undertest}$ equal $T_{undertest,1}$ and once with $T_{undertest}$ equal to $T_{undertest,2}$.
(5) If no alternative execution path exists and $T_{undertest}$ contains exactly one transition, then this transition is added to the list of undecomposable transitions.
(6) Once all alternative executions paths for $P_i$ are computed (including all forks), if there were one or more transitions found to be undecomposable in the original set of transitions under test (computed at Step 1), delete all generated alternative execution paths and go back to Step 1.

In order to understand step 6, let us emphasise that dividing alternative execution paths in step 4 increases substantially test effort (more parallel executions, more feature switches, longer testing time in general). However, one undecomposable transition among the transitions under test $T_{undertest}$ is enough for the generation of an alternative execution path to be impossible for $T_{undertest}$ to be split in step 4. To avoid having one undecomposable transition compromising an entire set of transitions under test, step 6 will instead regenerate all generated alternative execution paths if there were undecomposable transitions in the original set.

The second step takes a pair of consecutive test configurations (the original execution path) and a list of transitions under test for this pair of configurations, as defined by the first step. Its role is to generate an alternative execution path that excludes the transitions under test with the same starting and end point as the pair of test configurations. This optimisation problem is not trivial. We must define intermediate configurations whose feature assignment does not violate the model's constraints while excluding particular sequence of feature values. Additionally, there may be multiple correct alternative execution paths. Our goal is then to minimise the number of intermediate configurations in order to reduce the test effort needed to use our test oracle.

This problem can be seen as a multi-step configuration process in the context of evolving feature model configurations [29]. A formal model of multi-step configuration can be mapped to constraint satisfaction problems (CSPs). Most importantly, solutions to these SPL configuration problems can be automatically derived with a constraint solver, such as a solver that uses satisfiability modulo theories (SMT solver). Finally, from empirical results, CSP-based reasoning techniques have been shown to scale to SPL models with hundreds of features and multiple configuration steps.

We now focus on the generation of an alternative execution path. We will use a SMT solver in order to define $N$ intermediate configurations (from $C_1$ to $C_N$) between a given start configuration ($C_0$) and final configuration ($C_{N+1}$), where $C_0$, $C_{N+1}$ and $N$ are hyper-parameters.

The assignment of values for each intermediate configuration $C_i$ must be valid according to the feature model $FM$. Solvers have long shown that they can reason upon the relationships in a feature model [3, 24]. A feature model is usually composed of *mandatory*, *optional*, *or*, *alternative*, *exclusion*, or *implication* relationships. Note that any constraint that the SMT solver can reason upon could be accepted as a constraint in the feature model $FM$ (e.g., *if-else* statements or complex propositional logic). The following statement must hold true for a path between a start and final configuration to be valid, that is, the assignment of values of each $C_i$ respects all constraints in FM:

$$\forall i \in (0, 1, ..., N+1), FM(C_i) = 1 \tag{6}$$

Note that Equation 6 also verifies that the start and final configurations respect the feature model constraints. In our case, if the input to our test oracle is a test suite generated by our CTT approach, then $i = 0$ and $i = N + 1$ can be skipped for optimisation as those configurations are known to be valid.

We must also define a set of constraints in order to exclude a set of particular transitions $T$. We use Definition 1 (Section 5, definition of the coverage criterion for a transition) to prevent coverage of all transitions in $T$. If $C_i$ contains the opposite values associated to the transition, then $C_{i+1}$ cannot complete the coverage (Equation 7). If $C_{i+1}$ contains the assignment of values associated to the transition, then $C_{i+1}$ also cannot complete the coverage (Equation 8). More formally, for an alternative execution path, that excludes particular transitions, to be valid the following statements must hold true:

$$\begin{gathered}\forall i \in (0, 1, ..., N), \forall \mid v_1, ...v_n \mid \in T : \\ (\{\neg v_1, ...\neg v_n\} \subseteq C_i) \implies (\{v_1, ...v_n\} \nsubseteq C_{i+1})\end{gathered} \tag{7}$$

$$\begin{gathered}\forall i \in (0, 1, ..., N), \forall \mid v_1, ...v_n \mid \in T : \\ (\{v_1, ...v_n\} \subseteq C_{i+1}) \implies (\{\neg v_1, ...\neg v_n\} \nsubseteq C_i)\end{gathered} \tag{8}$$

Let us simplify these equations to CNF (conjunctive normal form) formulas and show that Equation 7 and Equation 8 are equivalent. First, we define an atomic variable that represents whether a value $v$ is in a configuration $C_i$ for easier notations in Equation 9.

$$\forall i \in (0, 1, ..., N), \forall v \in FM : (v \subset C_i) \Leftrightarrow C_{v,i} = 1 \tag{9}$$

Equation 10 eliminates propositional logic and integrates the easier notations, then uses De Morgan's law to simplify negations for Equation 7.

$$\begin{gathered}\forall i \in (0, 1, ..., N), \forall \mid v_1, ...v_n \mid \in T : \\ (\{\neg v_1, ...\neg v_n\} \subseteq C_i) \implies (\{v_1, ...v_n\} \nsubseteq C_{i+1}) \\ \Leftrightarrow \\ \neg(\neg C_{v_1,i} \wedge ... \wedge \neg C_{v_n,i}) \vee (\neg C_{v_1,i+1} \vee ... \vee \neg C_{v_n,i+1}) \\ \Leftrightarrow \\ C_{v_1,i} \vee ... \vee C_{v_n,i} \vee \neg C_{v_1,i+1} \vee ... \vee \neg C_{v_n,i+1}\end{gathered} \tag{10}$$

We now do the same for Equation 8 in Equation 11 (elimination of conditional logic, use of easier notations then De Morgan's laws). The logical operator *OR* is commutative. With this property, we can affirm that the results are the same as in Equation 10, demonstrating their equivalence.

$$\begin{gathered}\forall i \in (0, 1, ..., N), \forall \mid v_1, ...v_n \mid \in T : \\ (\{v_1, ...v_n\} \subseteq C_{i+1}) \implies (\{\neg v_1, ...\neg v_n\} \nsubseteq C_i) \\ \Leftrightarrow \\ \neg(C_{v_1,i+1} \wedge ... \wedge C_{v_n,i+1}) \vee (\neg(\neg C_{v_1,i}) \vee ... \vee \neg(\neg C_{v_n,i}) \\ \Leftrightarrow \\ \neg C_{v_1,i+1} \vee ... \vee \neg C_{v_n,i+1} \vee C_{v_1,i} \vee ... \vee C_{v_n,i}\end{gathered} \tag{11}$$

Even more important than simplifying the exclusion of transitions, these equivalences show that our constraints can be represented through CNF formulas. While using SMT solvers can reason upon more complicated formulas, using CNF formulas has been shown to be much more efficient resource-wise [22].

Finally, we define a minimisation objective to find the shortest alternative execution path, i.e. find a configuration $C_k$ (where $k \leq N$) equal to the final configuration $C_{N+1}$ as soon as possible. Our minimisation objective is then the number of intermediate configurations different from the final configuration.

Among $p$ possible paths $P_1, ...P_p$, the value of $objective_i$ is the objective function of $P_i$. First, Equation 12 determines a series of equal state variables $E_j$ used in Equation 13 to define the minimisation objective $objective_i$:

$$\begin{gathered}\forall j \in (1, ..., N), (C_j = C_{N+1}) \implies E_j = 0 \\ (C_j \neq C_{N+1}) \implies E_j = 1\end{gathered} \tag{12}$$

$$objective_i = \sum_{j=1}^{N} E_j \tag{13}$$

The final path is thus the one associated to the lowest objective:

$$objective_{final} = min_i(objective_i) \tag{14}$$

## 8 VALIDATION

We proposed a test generation algorithm that combines both interaction and transition testing as well as a test oracle for error detection. This section starts by answering RQ3: *What is the test effort of adding transition coverage to interaction coverage?* We first analyse the impact on test effort of the three improvements of our algorithm on our illustrative case study (Section 8.1). We then generalise by using the S.P.L.O.T. [23] repository (Software Product Lines Online Tools). This repository allows for more generalised

hyper-parameter tuning (Section 8.2) and a more fine-grained comparison of the performance of both CIT and CTT (Section 8.3). The second part of this section then uses this repository to answer RQ5: *What is the test effort of using a test oracle to detect behavioural transition errors?* (Section 8.4).

Our tool was implemented in Python using the incremental SAT solver glucose3 [1] for the generation algorithm and the SMT solver z3 [2] for the test oracle. Readers interested in reproducing the results of this section (Table 5, Table 6, Table 7 and all related conclusions) can access the following Github repository: https://github.com/PierreMartou/JSS2024-ReplicationPackage.

## 8.1 Improvements analysis

To analyse the impact of each improvement on performance *and* on each other, Table 5 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 5: 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 (both size and cost) 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. In other words, the generation algorithm prefers covering a lot of interactions while covering few transitions, even though some interactions are invertible and will be anyway covered by future transition coverage (Definition 3). 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 average performance metrics and standard deviation. The comparative heuristic is more suited for performance stabilisation (as shown by the reduced standard deviation).

[2] https://github.com/Z3Prover/z3

As for the cost, it varies proportionally to the size. A possible outcome was that even if the size of generated test suites were reduced, the overall number of transitions would remain the same 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%.

## 8.2 Hyper-parameter tuning

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 will now tune our hyper-parameters for a large number of models using S.P.L.O.T. [23], 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 CTCR, is the ratio of the number of features in the cross-tree constraints to the number of features in the feature tree [24]. 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. However, we assume that improving performance metrics for a large and representative set of feature models would also improve the particular case of dynamically adaptive software systems.

The previous section analysed the impact of our improvements with fixed weight values (0.5 and 0.3 for the look-ahead and comparative mechanisms, respectively), as these values yielded the best results for our case study. However, these values could be sub-optimal when analysing a large number of different feature models. We now use the repository S.P.L.O.T. to find optimal values in most of the cases. For this section, we randomly sampled 260 models containing between 10 and 50 features in S.P.L.O.T..

First, we look at the impact of changing the look-ahead weight (Improvement 2). We compute the average size of generated test suites at different values for the look-ahead (between 0.1 and 1, every 0.1). For this experiment, we set the comparative weight at a fixed value of 0.5. A look-ahead weight of 0 is equivalent to no look-ahead mechanism and will be analysed in the next subsection and is thus omitted here. The impact on the size and cost of generated test suites is shown in Figure 4.

We can observe that the impact on size is relatively marginal as long as the weight is between 0.2 and 0.9. As expected, a value of 1 yields worse results since the algorithm will value equally present and future coverage, potentially continuously preparing transition coverage without ever actually covering transitions.

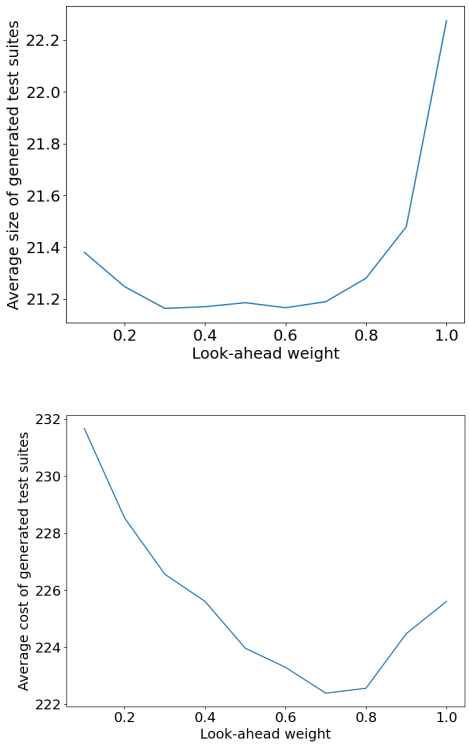


**Figure 4: Average size and cost for 260 models with different look-ahead weights.**

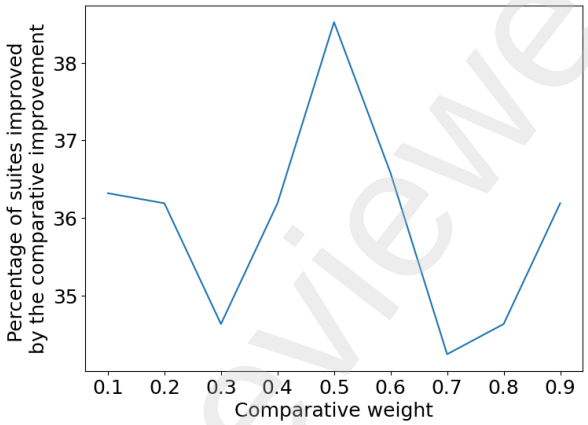


**Figure 5: Percentage of feature models (among 260) that benefit from the comparative improvement.**

As for the cost, higher values for the look-ahead weight tends to lower the cost as long as it is less than 0.8. The look-ahead mechanism is comparable to a smoothing mechanism: by considering both present and future coverage, it avoids aggressive coverage which (de)activate a lot of features but are not useful for future coverage, hence leading to non-optimal choices. In conjunction with the impact on average size, we decided on 0.7 as the optimal value and will be kept for all following computations.

The impact of the comparative improvement is more nuanced. While its aim is to stabilise the results of the generation algorithm (i.e. smaller standard deviation), it can also marginally increase the average size of generated test suites depending on the feature model. To find an optimal value, we show in Figure 5 the percentage of test suites where, on average, this comparative weight value finds smaller test suites compared to not using the Improvement 3. The higher percentage is the most interesting, and a value of 0.5 will be kept for all following computations. The following subsection highlights how the comparative mechanism (Improvement 3) stabilises the generation of test suites by reducing the standard deviation of their sizes and costs.

We now analyse the impact of different improvement combinations on various and larger models, using the public feature model repository S.P.L.O.T. [23] in Table 6. **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.

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). This repository contains a large number of feature models with few constraints. To assure the quality of the results, our uniform sampling has a bias towards feature models with more complicated constraints to compensate the larger number of feature models with fewer constraints. More specifically, we sampled in priority feature models with an *alternative* relationship between features or at least one cross-tree constraint. 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.

Without improvements, the generation algorithm can generate a large number of test configurations because of few specific transitions hard to cover due to complex constraints. To represent this issue, we put a limit on the maximum number of test configurations before a generation is considered to time-out. This limit is specific to each feature model and is equal to five times to average size of a CIT-generated test suite for this feature model. Time-out rate (percentage of generations which time-out) are in parentheses. Note that a generation which time-outs is not counted in the metrics.

First, we observe that the look-ahead heuristic (Improvement 2) is vital to the scalability and validity of CTT: it substantially decreases time-outs, which are prevailing at higher number of features. Without it, our generation algorithm cannot scale. The look-ahead heuristic (Improvement 2) has thus the largest impact on performance, and is further stabilised by the invertible interaction filter (Improvement 1) and comparative weight heuristic (Improvement 3).

**Table 6: Analysis of the size, cost of test suites generated through transition testing with different improvement combinations. All metrics are averaged over 5 generated test suites. Time-out rate is in parentheses.**

| #features | Qty | Size ± standard deviation (time-out rate) | | | | Cost ± standard deviation (time-out rate) | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | CTT 0 | CTT 1 | CTT 1&2 | CTT 1&2&3 | CTT0 | CTT 1 | CTT 1&2 | CTT 1&2&3 |
| 10-19 | 134 | 20.1±5.9 (39.8%) | 22.8±4.5 (23.4%) | 15.1±0.4 (0.7%) | 15.1±0.4 (0.0%) | 94.0±15.8 (39.8%) | 103.4±12.8 (23.4%) | 90.5±2.2 (0.7%) | 90.8±2.7 (0.0%) |
| 20-29 | 51 | 25.6±5.7 (57.5%) | 32.5±4.5 (53.6%) | 19.7±0.7 (0.0%) | 19.7±0.7 (0.0%) | 176.6±28.0 (57.5%) | 207.0±23.4 (53.6%) | 189.5±4.8 (0.0%) | 189.3±4.8 (0.0%) |
| 30-39 | 34 | 38.1±6.1 (71.6%) | 38.5±5.1 (68.6%) | 27.0±0.9 (0.0%) | 26.6±0.9 (0.0%) | 390.9±46.6 (71.6%) | 401.5±38.9 (68.6%) | 343.9±9.5 (0.0%) | 341.4±9.6 (0.0%) |
| 40-49 | 38 | 45.6±6.6 (88.6%) | 86.5±7.5 (79.8%) | 39.2±2.1 (0.9%) | 40.0±1.2 (0.0%) | 453.7±62.4 (88.6%) | 969.3±81.1 (79.8%) | 620.2±26.9 (0.9%) | 627.8±15.0 (0.0%) |
| 50-69 | 25 | 111.3±4.3 (88.0%) | 94.9±3.9 (86.7%) | 44.6±1.7 (0.0%) | 44.3±1.4 (0.0%) | 1766.8±53.2 (88.0%) | 1532.3±61.8 (86.7%) | 979.6±25.7 (0.0%) | 973.6±22.0 (0.0%) |
| 70-99 | 17 | N/A (100.0%) | 283.6±6.8 (90.2%) | 59.5±3.6 (2.0%) | 59.8±2.2 (0.0%) | N/A (100.0%) | 5046.0±148.2 (90.2%) | 1708.3±147.1 (2.0%) | 1721.7±56.8 (0.0%) |

As expected, the comparative mechanism (Improvement 3) has less impact on average size and cost than it has on standard deviation. All categories combined, CTT 1&2&3 has marginally higher sizes of test suite (0.8% higher) but reduces standard deviation by 20% compared to **CTT 1&2**. The improvement is even more significant when looking at average cost: all categories combined, the Improvement 3 marginally increases the average cost by 0.7% but reduces the standard deviation by 39.8%.

In conclusion, using all improvements (CTT 1&2&3) yields the best results for transition testing. They enable our testing approach for any number of feature by preventing time-outs, and greatly stabilise the average size and cost of generated test suites. From now, **CTT 1&2&3** (CTT with all improvements) will be used by default and shortened to **CTT**.

## 8.3 Comparison of CIT and CTT

We now validate our testing approach using the same public feature model repository S.P.L.O.T. [23]. Our goal is to compare interaction testing (**CIT**) to transition testing to assess scalability of our generation approach with regards to CIT (which was proven to have logarithmic growth [5]).

We also compute the transition coverage when using CIT (**T. cov.**), i.e. the percentage of transitions covered, similarly to the analysis 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 **diss**imilarity 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. The results of our analysis using S.P.L.O.T. is shown in Table 7.

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 1.88 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 ~0.956, 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 similar to 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 90%, 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 90% 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 90% 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, with higher variance. Compare this to our transition testing approach which is less than double the size and cost compared to standard CIT and achieves full transition coverage in a single test suite.

**Table 7: Analysis of the size, cost and transition coverage of test suites generated through CIT with different reordering and CTT. All metrics are averaged over 5 generated test suites.**

| #features | Qty | Size | | Cost | | | Transition coverage | |
|---|---|---|---|---|---|---|---|---|
| | | CIT | CTT | CIT Diss | CIT Cost | CTT | CIT Diss | CIT Cost |
| 10-19 | 134 | 8.3 | 15.1 | 46.0 | 24.4 | 90.8 | 69.0 | 23.1 |
| 20-29 | 51 | 11.1 | 19.6 | 102.6 | 54.9 | 189.3 | 71.0 | 29.2 |
| 30-39 | 34 | 14.3 | 26.6 | 186.8 | 92.0 | 341.6 | 76.1 | 29.4 |
| 40-49 | 38 | 19.9 | 40.2 | 333.7 | 169.6 | 628.2 | 80.7 | 35.4 |
| 50-69 | 25 | 22.4 | 44.2 | 529.4 | 276.5 | 972.4 | 83.1 | 39.9 |
| 70-99 | 17 | 32.9 | 60.1 | 1160.4 | 619.4 | 1725.9 | 87.0 | 50.8 |

## 8.4 Test oracle performance and cost

For our experiments, we used the SMT solver z3 (https://github.com/Z3Prover/z3). The goal of this section is to experimentally validate the feasibility of our test oracle. Similarly to the generation algorithm for CTT, the test effort needed to use our test oracle must scale logarithmically in the number of features in order to be usable.

Our test oracle generates alternative executions paths in order to determine whether an error occurs or not during execution. Several aspects of our test oracle can act as performance metrics, and will be more or less important resource-wise, depending on the system under test. We now describe each evaluated metrics (and their **abbreviation**):

(1) **#intermediate** The number of intermediate configurations generated by the test oracle. Similarly to the size of generated test suites, we want as few intermediate configurations as possible.
(2) **#paths** The total number of alternative execution paths. Each alternative execution path results in an internal state that must be compared to the internal state resulting from the original execution path, which can be resource-expensive.
(3) **Cost** The total number of feature (de)activations in the alternative execution paths. Similarly to the cost of generated test suites, we want the cost to be as low as possible.
(4) **Undecomposable** The rate of undecomposable transitions compared to the total number of valid transitions, in percentage. Undecomposable transitions can trigger unexpected behaviour that will not be detected by our test oracle.

Before performance evaluation, we must find the optimal value for $N$, a hyper-parameter of our test oracle. $N$ determines the maximum number of intermediate configurations between start and final configurations in alternative execution paths. Paradoxically, increasing $N$ (the possible number of intermediate configurations) actually lowers the total number of intermediate configurations our test oracle generates. This is illustrated in Figure 6. This can be explained by the fact that higher $N$ allows the detection of more transitions in a single and longer alternative execution path (i.e. more transitions can be excluded in the same alternative execution path). One longer execution path of 5 intermediate configurations is more efficient than two execution paths with 3 intermediate configurations each. Increasing $N$ results in an increase in overall efficiency and decrease in the number of intermediate configurations. However, increasing $N$ also has an impact on computational efforts and has diminishing returns on efficiency. We advise stopping at a value of $N = 6$, and will use this value for further computations.

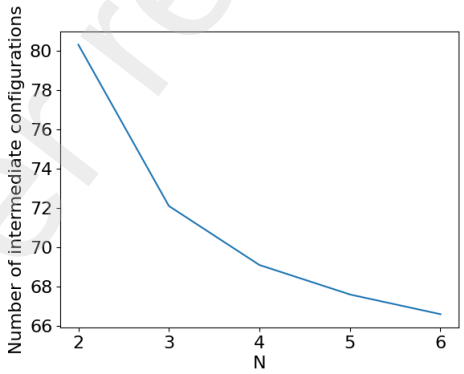


**Figure 6: Evolution of the total number of intermediate configurations in alternative execution paths for different values of $N$**

Table 8 reports all metrics previously described from smaller to bigger feature models. The sampling is the same as in Section 8.3. As before, for clarity, we aggregated these feature models according to their number of features and indicated the categories in **#features** and number for each category in **Qty**. All metrics are averaged over 3 runs.

First, let us discuss the average number of intermediate configurations and number of paths and their relation to the size of CTT-generated test suites reported in Table 7. We observe an average factor of 3.5 (resp. 1.7) between the size of CTT-generated test suites and the size of alternative execution paths (resp. number of alternative executions paths) generated by our test oracle. More importantly, the Pearson correlation coefficient between the size of CTT-generated test suites and size of alternative execution paths (resp. number of alternative execution paths) at a given feature model size is of 0.91 (resp. 0.94), where 1 means linear correlation. While it is lower than the relation between CIT and CTT, these high correlations still hints that our test oracle also grows logarithmically in the number of features. We also believe that it could be further improved with more suited heuristics at higher number of features.

While scalability is important, we also wanted to assess the impact of undecomposable transitions. The rate of undecomposable

**Table 8: Analysis of the average number of intermediate configurations, alternative execution paths, parallel paths and cost when using our test oracle with $N = 6$, as well as rate of undecomposable transitions.**

| #features | Qty | # intermediate | # paths | Cost | Undecomposable |
|---|---|---|---|---|---|
| 10-19 | 134 | 26.1 | 15.1 | 313.6 | 3.6% |
| 20-29 | 51 | 45.0 | 22.4 | 807.8 | 2.1% |
| 30-39 | 34 | 67.8 | 31.4 | 1593.7 | 2.9% |
| 40-49 | 38 | 112.4 | 50.0 | 3314.9 | 1.6% |
| 50-69 | 25 | 145.2 | 62.0 | 5564.4 | 1.9% |
| 70-99 | 17 | 230.7 | 86.1 | 11845.7 | 0.4% |

transitions compared to the total number of valid transitions is very low even at low number of features (3.6%). We observe that this rate substantially decreases as the number of features grows. Undecomposable transitions can be seen as a set of features which only allow one single path between a start and end configurations, hence making it impossible to find alternative paths. At higher number of features, this becomes increasingly uncommon because of the higher number of possible configurations (and hence possibilities in alternative paths), which explains the decreasing rate. We conclude that the weakness of our test oracle, the undecomposable transitions, concerns very few transitions and becomes irrelevant when increasing the number of features.

## 9 THREATS TO VALIDITY

The focus of this paper was illustrating behavioural transition errors and providing a feasible solution for covering them on feature models with less than 100 features, then defining a test oracle to detect these errors. Our proof of logarithmic growth for CTT and test oracle 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 [5]). 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. We still need to evaluate it in practice on realistic dynamically adaptive systems.

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 (generation algorithm and test oracle) 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 for the generation part and SMT solver calls for the test oracle 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.

## 10 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 full transition coverage easily, and we would need to combine more than ten test suites together to find all potential behavioural transition errors (RQ1), resulting in enormous test effort.

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 of generated test suites. Moreover, the original algorithm can hit a dynamic time-out (set to five times the size of test suites generated by CIT for each feature model) for many of the analysed feature models, ranging from a quarter of small feature models to nearly all models when approaching 100 features. In these scenarios, the initial algorithm keeps producing new test configurations while failing to achieve full transition coverage until it hits the time-out limit. Our optimisations effectively eliminate this issue (RQ2). Finally, we validated its logarithmic growth in the number of features. CTT 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.

We further developed our testing approach with a test oracle for transition errors. In practice, just like it is infeasible to test every single valid configuration, it is also infeasible to define what is the correct behaviour for every single valid configuration. To solve this issue, we proposed a test oracle which can detect unexpected behaviour linked to erroneous transitions without any prior knowledge of the system's expected behaviour (RQ4). We evaluated the test effort needed to use our test oracle, compared it to CTT and showed it was also correlated, enabling its use in practice.

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 [9, 10] to such a set [21]. 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, combinatorial 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 [21], early fault detection [30], or prioritisation of frequent behaviour [8]. 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 [13]), 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 [14] and Kowal [16] 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 [21].

## 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] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Information systems* 35, 6 (2010), 615–636.

[4] Cagatay Catal and Deepti Mishra. 2013. Test case prioritization: a systematic mapping study. *Software Quality Journal* 21, 3 (2013), 445–478.

[5] 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.

[6] 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.

[7] Ismayle 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.

[8] 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.

[9] Benoît Duhoux. 2022. *Feature-Based Context-Oriented Software Development*. Ph. D. Dissertation.

[10] 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.

[11] 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.

[12] 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).

[13] 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.

[14] 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.

[15] 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.

[16] Matthias Kowal, Sofia Ananieva, and Thomas Thüm. 2016. Explaining Anomalies in Feature Models. *ACM SIGPLAN Notices* 52, 3 (2016), 132–143.

[17] 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.

[18] 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.

[19] 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.

[20] Pierre Martou, Benoît Duhoux, Kim Mens, and Axel Legay. 2024. Combinatorial Transition Testing in Dynamically Adaptive Systems. In *Proceedings of the 18th International Working Conference on Variability Modelling of Software-Intensive Systems* (, Bern, Switzerland,) *(VaMoS '24)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3634713.3634726

[21] Pierre Martou, Kim Mens, Benoît Duhoux, and Axel Legay. 2022. Test scenario generation for feature-based context-oriented software systems. *JSS* (2022), 111570.

[22] Gabriele Masina, Giuseppe Spallitta, and Roberto Sebastiani. 2023. On CNF conversion for disjoint SAT enumeration. In *26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik.

[23] 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.

[24] 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.

[25] Changhai Nie and Hareton Leung. 2011. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)* 43, 2 (2011), 1–29.

[26] 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.

[27] Ronny Seiger and Thomas Schlegel. 2012. Test modeling for context-aware ubiquitous applications with feature petri nets. In *MODIQUITOUS*. Citeseer.

[28] Teng Wang, Zhouyang Jia, Shanshan Li, Si Zheng, Yue Yu, Erci Xu, Shaoliang Peng, and Xiangke Liao. 2023. Understanding and detecting on-the-fly configuration bugs. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*.

[29] Jules White, José A Galindo, Tripti Saxena, Brian Dougherty, David Benavides, and Douglas C Schmidt. 2014. Evolving feature model configurations in software product lines. *Journal of Systems and Software* 87 (2014), 119–136.

[30] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *STVR* 22, 2 (2012), 67–120.