

# Testing Tool for Combinatorial Transition Testing in Dynamically Adaptive Software Systems

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  
Nexova

**Abstract**—Dynamically adaptive software systems exhibit an exponential number of configurations, posing significant testing challenges. Combinatorial Interaction Testing (CIT) and Combinatorial Transition Testing (CTT) enable concise test suite generation to cover valid pairs of features and transitions. Detecting behavioural transition errors in such test suites with full transition coverage still remains complex and time-consuming. We propose a modular and adaptable testing tool implementing state-of-the-art CTT. It generates test suites using either CIT or CTT and integrates a dedicated test oracle for efficient detection of behavioural transition errors, with minimum user effort.

**Index Terms**—testing tool, combinatorial testing, transition testing, test oracle, interaction testing, dynamically adaptive software systems, feature modelling

## I. INTRODUCTION

A common approach to modelling dynamically adaptive software behaviour is through feature models [1], where a configuration represents the set of active features. Despite being highly constrained, these models still face a combinatorial explosion of configurations, making exhaustive testing impractical. *Combinatorial Interaction Testing* (CIT) addresses this by generating concise yet representative test suites covering all valid combinations of features.

CIT allows to find nearly all errors in a wide range of systems, with sufficiently high levels of coverage, where  $n$ -wise interaction testing ensures that all combinations of features of size  $n$  are covered [2], [3]. Its applicability to feature-based context-oriented software systems, a kind of dynamically adaptive software systems [4], has been studied [5].

While CIT is guaranteed to cover all *behavioural interaction errors* of a certain size, it may miss specific errors known as *behavioural transition errors* [6]. Whereas the former are triggered by the presence of specific features inside a configuration, the latter only appear when particular feature *transitions* occur in the system. *Combinatorial Transition Testing* (CTT) extends CIT to address such errors by generating test suites that cover all feature transitions of size  $n$ .

After generating representative test configurations through CIT and CTT, the actual system behaviour under these configurations is not yet tested. A test oracle can be used to verify if the system behaves as expected for each configuration [7]. Given a suite of test configurations, a perfect test oracle can detect all errors provoked by the execution of the test suite. Using such an oracle,  $n$ -wise CIT and CTT are guaranteed to

detect all behavioural interaction and transition errors (of size  $n$  or less).

Constructing a perfect test oracle is complex and impractical due to the combinatorial explosion of configurations and their expected behaviour in dynamically adaptive software systems. Instead, a dedicated test oracle for detecting behavioural transition errors has been proposed [7], requiring no prior knowledge of the expected behaviour. It merely compares the original execution path with generated alternative paths. When equivalent test configurations yield divergent system states for different execution paths, this signifies that some transition in the execution paths caused inconsistent behaviour.

As designing tests for dynamically adaptive systems is complex and time-consuming, this paper presents a dedicated testing tool that integrates CIT, CTT, and the test oracle. Even though we use feature models to showcase the testing tool's functionalities, the tool accepts any model that can be reduced to a set of Conjunctive Normal Form (CNF) constraints.

To make it easier to connect real-world dynamically adaptive software systems and research, our proposed tool implements and optimises state-of-the-art research on Combinatorial Transition Testing [7]. It also serves for research dissemination and experiments (e.g. comparison with alternative testing techniques), and is sufficiently modular and adaptable for those who want to refine some strategies.

The remainder of this paper is structured as follows. Section II describes background material needed to understand our testing tool, exemplified with a simple dynamically adaptive software system. Section III presents the different functionalities of our tool and Section IV concludes and outlines future steps to complete it.

## II. BACKGROUND

This section explains in more detail CIT (cf. II-A), CTT (cf. II-B) and the test oracle (cf. II-C) used in our testing tool.

### A. Combinatorial Interaction Testing

Combinatorial Interaction Testing (CIT) is a system-level testing technique that ensures complete test coverage of software product lines. To cope with the explosion of possible configurations of these systems, CIT assumes that most errors stem from interactions between features [2], [3]. CIT thus generates a pertinent yet tractable set of configurations (or

test suite) that cover all possible interactions of features. A test configuration is said to cover a given set of features with values  $v_1, \dots, v_n$  if it includes this particular set of values.

Executing a test suite on a dynamically adaptive system involves consecutive feature (de)activations, which can be resource-intensive. To optimize real-world execution of such test suites, test configurations are often reordered (prioritised) to reduce simulation efforts [5] or enable early fault detection [8]. Minimizing changes between consecutive configurations reduces resource usage during simulation, while prioritizing error-prone configurations ensures faster fault detection, especially when interruptions to the testing process are expected.

During their lifetime, dynamically adaptive software systems may evolve. Additional features can be introduced or existing constraints changed to meet new requirements [9]. Such evolutions can cause existing tests to become invalid or obsolete (incomplete interaction coverage). Generating a new test suite and the implementation of new tests from scratch can be time-consuming. Instead, test suite augmentation aims at maximising reuse of existing test suites [5], [10]. In the context of CIT, test suite augmentation modifies a previously-generated test suite, with minimal changes, to recover a valid test suite with full interaction coverage.

### B. Combinatorial Transition Testing

Combinatorial Transition Testing (CTT) aims to identify faulty transitions, where a transition is the simultaneous (de)activation of a given set of features. Similar to the coverage of interactions in CIT, the goal of CTT is to generate a test suite that covers all possible transitions, i.e. where all valid transitions of size  $n$  appear at least once in the test suite. CTT is particularly relevant for dynamically adaptive systems, where such transitions are both frequent and potentially error-prone, and because CIT fails to consistently find all errors caused by faulty transitions [11].

To implement pairwise CTT, Martou et al. proposed a greedy row-by-row generation algorithm with an additional transition coverage objective [6]. Adding the transition objective to the greedy algorithm in a naive implementation led to exponential scaling of the test suite size in the number of features. To address that problem, three optimisations were integrated into the naive iteration. The first optimisation, *interaction filtering*, prevents non-optimal bias towards interaction coverage in the generation of the test suite. The second optimisation, the *look-ahead mechanism*, significantly enhances the greedy heuristic of the algorithm by encouraging it to anticipate future transitions. Finally, the *comparative mechanism* optimisation prioritises features with high difference in potential between values rather than those whose values are equally significant to prevent non-optimal decisions of the greedy heuristics. With these optimisations, for a given system, the sizes of CIT- and CTT-generated test suites have been shown to correlate linearly, making CTT an effective supplement in dynamically adaptive software systems [6].

### C. Test oracle

After having generated a test suite with CIT or CTT, we still need to detect behavioural errors. Implementing tests manually is resource intensive and can lead to the human oracle cost problem [12], [13]. A test oracle is therefore essential to automatically detect errors. But creating a perfect oracle is challenging due to non-determinism, incomplete specifications, system complexity, or combinatorial explosion of the state space. To tackle these different challenges, many approaches rely on approximations or heuristics, which may cause false positives or negatives [14], [15].

In transition testing, the test oracle's scope can be narrowed down. A key property of dynamically adaptive systems is consistency, meaning that the effect of a specific change to the system should remain the same regardless of its previous state [16]. To verify consistency, a partially automated test oracle can be achieved by relying on differential testing principles [7]. Given a test suite of  $k$  test configurations, the oracle generates alternative execution paths that all possess the same  $k$  test configurations, in the same order, but with additional configurations inserted in between. These alternative paths exclude specific transitions under test that will occur in the original path only. After executing both the original and alternative test suites, the oracle checks for consistency. All  $k$  configurations must exhibit identical behaviour, regardless of the system's previous history. If an inconsistency is detected, the test oracle identifies the responsible erroneous transition.

## III. FUNCTIONALITIES

Having covered all concepts needed to understand our testing tool, we now present its main functionalities progressively, from CIT, via CTT, to the test oracle. The tool is available on GitHub.<sup>1</sup>

### A. Generation for Combinatorial Interaction Testing

The tool's first functionality is CIT *generation*. It produces a test suite that covers all pairwise combinations of features of a system [5]. Additional functionalities are test suite *reordering* to reduce test effort and *test suite augmentation* to easily ensure full coverage when expanding a system.

```
line = parent / relation / children
parent = "feature"
relation = MANDATORY | OPTIONAL | OR |
ALTERNATIVE
children = "feature" | "feature" - children
```

Fig. 1. EBNF Grammar for feature models. Italics are terminal symbols.

1) *Generation* of a test suite requires as input a description of the system's features, following the EBNF grammar of Figure 1. We also accept input in the popular DIMACS format<sup>2</sup>, which describes a set of CNF constraints. A feature model

<sup>1</sup> <https://github.com/PierreMartou/CIT-SAT-Implementation>

<sup>2</sup> <https://jix.github.io/varisat/manual/0.2.0/formats/dimacs.html>

can be exported to DIMACS format with the FeatureIDE<sup>3</sup> modelling tool. An example of such a DIMACS file can be found in *Data/ToolExample/DIMACSexample.cnf*.

To generate a test suite using CIT, we have to use the method *computeCITSuite*. An example of its usage is illustrated in *ToolExample/CITUsage.py*. It allows to store and retrieve a generated test suite using the file path *fpath*, for a specific system *s*, with some utility parameters, such as the *iteration* that defines the current iteration of this system, the number of *candidates* used by the greedy algorithm and if we have to *recompute* the test suite. Table I summarises the arguments of this method. The output of this method is the test suite represented as a list of all configurations, where a configuration is a dictionary whose keys are the system's features and whose values are positive (resp. negative) integers to indicate an activated (resp. deactivated) feature.

The current implementation uses the renowned SAT solver Glucose3 [17] of the pysat.solvers API<sup>4</sup>. Due to the modularity of our object-oriented design, the SAT solver used can easily be swapped to a custom one, as long as it supports the satisfiability verification of a partial assignment of values.

2) *Reordering* test suites (cf. II-A) is often desired to minimise test effort or optimise test fault detection. Multiple reordering strategies are implemented in the tool, all obtainable through the *TestSuite* class. The first strategy is *cost optimisation*. It minimises the number of feature changes between consecutive configurations with a greedy reordering algorithm [5]. The *getMinTestSuite* method computes this rearrangement. Optimising early fault detection is done through the maximisation of dissimilarity between consecutive configurations through the Jaccard distance [18]. This can be done with the *getMaxTestSuite* method. Finally, one can generate a random ordering for comparison purposes with the *getRandomTestSuite* method.

3) *Test suite augmentation* provides a way to maximise reuse of a previously-generated test suite upon evolution of the system's features. Our tool implements, in the *TestsAugmentation* class, the test suite augmentation algorithm of Martou et al. [5]. Its inputs are the previous test suite and a definition of the new system (i.e. there is no need to keep track of the changes applied to the system explicitly). Its output is a description of what to change or add to the existing test suite.

Similarly to CIT, test suite augmentation follows the same modularity principle when using SAT solving, allowing for straightforward redirection to a custom solver.

### B. Generation for Combinatorial Transition Testing

The base input for test suite generation using CTT is the same as for CIT generation: a feature model formatted according to the EBNF grammar of Figure 1, or a description of the system in DIMACS format. As output, the generated test suite with full transition coverage is in the same representation as for CIT. An example of its usage is illustrated in *ToolExample/CTTUsage.py*.

Our tool implements the greedy algorithm and the three optimisations defined in Section II-B [7]. To generate a CTT test suite, the *computeCTTSuite* method is a useful wrapper function for all parameters related to the optimisations. All optimisations can be used independently from one another. The *interaction filter* optimisation can either be activated or deactivated. The *look-ahead mechanism* uses a weight that influences its impact, from 0 (no impact, equivalent to no optimisation) to 1 (strongest impact). By default, the weight is 0.7, a value found to be optimal for a large choice of feature models. The *comparative mechanism* also uses a weight ranging from 0 (no impact) to 1 (strongest impact), with a default value of 0.3. These parameters are summarised in Table II.

Since the optimisations are needed to achieve logarithmic growth in the number of features, the CTT generation algorithm can fail to complete full transition coverage in reasonable time if they are all disabled. By default, a fail-safe would stop generation after 1000 test configurations in a test suite.

### C. Test oracle

The test oracle's goal is to find discrepancies between alternative execution paths, to determine whether particular feature transitions are correct or not. This process involves three steps: generating alternative execution paths, running them on the system under test, and performing consistency checks. The test oracle is semi-automated in the sense that for the third step, one needs to provide a way for the test oracle to compare whether two states of the system are equivalent. To facilitate this, we propose a simple framework linking the system under test with the testing tool. This process is illustrated in *ToolExample/TestOracleUsage.py*, and the framework resides in *TestOracle/TestOracleExecutioner*.

The first step builds upon a given test suite to generate alternative execution paths. This is typically the test suite generated by the CTT implementation of the tool, but it also works with test suites with incomplete transition coverage, which can be useful to verify specific transitions quicker.

The second step involves executing both the original and alternative paths. For this, the framework provides a way to enable specific transitions on the system under test through the *executeTransitions* method, whose parameters are the features to activate and deactivate.

Finally, the test oracle needs a way to compare the system's behaviour for identical configurations and different past transitions. The *State* class, part of the framework, is responsible for creating a description of the system's behaviour (its state) at a given point of time, as well as to compare different states. The *storeState* method achieves the former and returns an object of any kind representing the state of the system. The *compareState* method takes two such objects and returns whether they are equivalent or not.

The objects returned by *storeState* or taken as input by *compareState* are deliberately unrestricted, as system behaviour can be described in various ways. It may capture internal system values, results from different test modules, or user

<sup>3</sup><https://featureide.github.io/>

<sup>4</sup><https://pysathq.github.io/docs/html/api/solvers.html>

TABLE I  
DESCRIPTION OF THE PARAMETERS OF THE COMPUTECTSUITE METHOD.

| Parameter  | Default value | Usage                                                                                                                                                                                                                                                                         |
|------------|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| fpath      | -             | File path to find the previously-generated test suite if it exists, or store the newly-generated test suite.                                                                                                                                                                  |
| s          | -             | A path to a formatted feature model in a text file (.txt file), or a path to a DIMACS file (.cnf file), or directly an instance of the <i>SystemData</i> class) representing the internal representation of a system.                                                         |
| iteration  | 0             | Allows to store multiple iterations of a test suite at the same path for the same system (e.g. for statistics purposes). Works as an identifier for the current iteration.                                                                                                    |
| candidates | 30            | Number of candidates in the greedy row-by-row generation algorithm.                                                                                                                                                                                                           |
| recompute  | False         | If this parameter is set to <i>False</i> , the tool will first try to recover a test suite present in the given file path, if there is one. If it is set to <i>True</i> , it will regenerate a new test suite, regardless of what is currently stored at the given file path. |

TABLE II  
DESCRIPTION OF THE ADDITIONAL PARAMETERS OF THE COMPUTECTSUITE METHOD.

| Parameter          | Default value | Usage                                                                                                             |
|--------------------|---------------|-------------------------------------------------------------------------------------------------------------------|
| interaction_filter | True          | Activates the <i>interaction filter</i> optimisation.                                                             |
| weight_lookahead   | 0.7           | Influences the impact of the <i>look-ahead mechanism</i> optimisation, from 0 (no impact) to 1 (maximum impact).  |
| weight_comparative | 0.3           | Influences the impact of the <i>comparative mechanism</i> optimisation, from 0 (no impact) to 1 (maximum impact). |
| limit              | 1000          | Limits generation to 1000 configurations before a fail-safe stops the generation.                                 |

interaction simulations. Both methods act as integration points for existing system modules analysing system behaviour.

#### IV. CONCLUSION AND FUTURE WORK

This paper presented an adaptable testing tool integrating combinatorial interaction testing (CIT) and combinatorial transition testing (CTT) to address the challenges of testing dynamically adaptive software. It can generate concise and efficient test suites with both full interaction and full transition coverage. Furthermore, the integration of a semi-automated test oracle facilitates error detection, effectively connecting test suite generation with its practical application. The tool provides a solution for researchers seeking to reproduce or utilise state-of-the-art CTT techniques in real-world contexts.

Future work will enhance the tool's utility and performance through real-world validation with around 60 participants over four months in an academic setting. Expanding beyond pairwise testing to higher-order CTT could improve error detection, while refining the test oracle with new ways to compare system behaviour could enhance inconsistency detection. For systems with states too complex to describe and store, integrating parallel execution may enable real-time comparisons. These improvements aim to strengthen the tool's applicability in both academia and industry.

#### REFERENCES

- [1] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Carnegie-Mellon University Software Engineering Institute, Tech. Rep., 1990.
- [2] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys (CSUR)*, vol. 43, no. 2, pp. 1–29, 2011.
- [3] D. R. Kuhn and M. J. Reilly, "An investigation of the applicability of design of experiments to software testing," in *27th Annual NASA Goddard*. IEEE, 2002, pp. 91–95.
- [4] B. Duhoux, "Feature-based context-oriented software development," Ph.D. dissertation, 2022.
- [5] P. Martou, K. Mens, B. Duhoux, and A. Legay, "Test scenario generation for feature-based contextualized software systems," *Journal of Systems and Software*, vol. 197, p. 111570, 2023.
- [6] P. Martou, B. Duhoux, K. Mens, and A. Legay, "Combinatorial transition testing in dynamically adaptive systems," in *Proceedings of the 18th International Working Conference on Variability Modelling of Software-Intensive Systems*, 2024, pp. 1–10.
- [7] —, "Combinatorial transition testing in dynamically adaptive systems: Implementation and test oracle," *Journal of Systems and Software*, vol. 221, p. 112260, 2025. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0164121224003042>
- [8] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *STVR*, vol. 22, no. 2, pp. 67–120, 2012.
- [9] T. Thum, D. Batory, and C. Kastner, "Reasoning about edits to feature models," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 254–264.
- [10] A. Bombarda, S. Bonfanti, and A. Gargantini, "On the reuse of existing configurations for testing evolving feature models," in *Proceedings of the 27th ACM International Systems and Software Product Line Conference-Volume B*, 2023, pp. 67–76.
- [11] P. Martou, B. Duhoux, K. Mens, and A. Legay, "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, 2023, pp. 100–104.
- [12] P. McMinn, M. Stevenson, and M. Harman, "Reducing qualitative human oracle costs associated with automatically generated test data," in *Proceedings of the First International Workshop on Software Test Output Validation*, 2010, pp. 1–4.
- [13] S. Afshan, P. McMinn, and M. Stevenson, "Evolving readable string test inputs using a natural language model to reduce human oracle cost," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 352–361.
- [14] R. M. Hierons, "Oracles for distributed testing," *IEEE Transactions on Software Engineering*, vol. 38, no. 3, pp. 629–641, 2011.
- [15] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2014.
- [16] T. Wang, Z. Jia, S. Li, S. Zheng, Y. Yu, E. Xu, S. Peng, and X. Liao, "Understanding and detecting on-the-fly configuration bugs," in *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, 2023.
- [17] G. Audemard, J.-M. Lagniez, and L. Simon, "Improving glucose for incremental SAT solving with assumptions: Application to mus extraction," in *International conference on theory and applications of satisfiability testing*. Springer, 2013, pp. 309–317.
- [18] A. B. Sánchez, S. Segura, and A. Ruiz-Cortés, "A comparison of test case prioritization criteria for software product lines," in *ICST '14*. IEEE, 2014, pp. 41–50.