Beyond Combinatorial Interaction Testing: On the need for transition testing in dynamically adaptive context-aware 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
ICTEAM, UCLouvain
Louvain-la-Neuve, Belgium
axel.legay@uclouvain.be

***Abstract*—Testing context-aware systems is hard due to the combinatorial explosion of possible contexts and corresponding features that such systems can adapt to dynamically. Combinatorial interaction testing (CIT) has been successfully applied to generate small yet representative test suites covering all valid combinations of pairs of contexts and features of such systems. Such test suites can then be used to verify that the system behaves correctly in most cases. We claim that CIT may miss certain relevant scenarios that exhibit certain kinds of errors that only occur for particular sequences of context or features (de)activations (transitions). To generate test suites that capture such errors as well, we propose to complement the CIT approach with a new technique called Combinatorial Transition Testing (CTT). The combination of CIT and CTT presents some challenges but provides a better coverage of scenarios to be tested to find relevant errors in dynamically adaptive context-aware systems.**

***Index Terms*—transition testing, interaction testing, context-aware systems, dynamically adaptive systems, feature modeling**

I. Introduction

Context-aware software systems are systems that are aware of their surrounding environment and can adapt their behaviour accordingly. Context-Oriented Programming (COP) [1] is a class of programming languages to build context-aware applications that, based on contextual information sensed from their surrounding environment, can select the most appropriate behaviour at runtime. *Feature-Based Context-Oriented Programming* (FBCOP) [2] is a particular variant of such languages that reconcile COP, Feature Modelling (FM) [3] and (Dynamic) Software Product Lines (DSPL) [4], [5], [6] into a single software development approach.

Much like software product lines, FBCOP systems can be seen as a family of systems of which the actual behaviour depends on the current context of use. Possible behaviours (or features) of the system are described in terms of a feature model and the possible contexts in terms of a context model, both using a feature diagram notation. Which features are active at runtime depends on which contexts are currently active. A valid configuration of the context model describes a particular context of use, and triggers the selection of features specific to those contexts through a mapping model of contexts to features.

In this paper, testing context-aware systems is searching for behavioural errors, i.e. discrepancies between the system's expected and observed behaviour. Even though contexts and features are closely linked through the mapping model, making the system highly constrained, in practice the amount of possible configurations still makes it impossible in practice to test them all.

Previous research showed that Combinatorial Interaction Testing (CIT) can be successfully applied to FBCOP systems to generate small yet representative test suites covering all valid combinations of pairs of contexts or features [7]. In addition CIT has a varying degree of coverage: *n*-wise interaction testing means that all combinations of size n will be covered. With sufficiently high coverage, CIT has been claimed to find nearly all errors in a majority of systems [8], [9], making it a widely popular technique.

It should be emphasised that CIT only produces relevant configurations to be tested. In practice, a tester still needs a good test oracle to actually find errors from these test configurations (i.e., a tool that determines whether a test passes or not). For the sake of simplicity, in the rest of this paper we assume to have a perfect test oracle. In other words, if a test configuration contains a situation (a set of features being (de)activated together) that may cause a certain error, then this error is said to be found by this test configuration.

The idea of CIT is that some faults can be exposed by testing interactions among the different features (and the contexts that trigger their selection). *Behavioural interaction errors* are caused by an interaction between specific features, which means that the simple presence of these features is enough to trigger a behavioural error. With a perfect oracle, n-wise CIT will necessarily find all behavioural interaction errors (corresponding to interactions of size n or less).

Nevertheless, on the basis that the system under test is now context-aware and dynamic, in this paper we claim that there is a certain kind of errors that CIT may not always be able to find. Indeed, in such a system different configurations will be applied consecutively to the system features at runtime. Hence,

errors could come not only from a particular configuration but could also be due to a particular transition between configurations. Such *behavioural transition errors* are caused by a transition of specific features, meaning that switching specific features at the same time may trigger a behavioural error in the resulting configuration.

In a survey on test case design for context-aware systems the idea of covering all transitions was categorised as "State Transition Testing" [10]. To the best of our knowledge, transition testing was limited to systems which explicitly represented transitions between states (e.g., using Petri Nets [11]). In this paper, we study transition testing on systems whose modelling is closer to SPL, with no explicit description of possible transitions. It follows that our CIT-derived approach proposes its own transition testing definition and coverage criterion.

Another approach, "Combinatorial sequence testing", covers all sequences of specific events in a system [12], [13]. When applied to our system, this approach could verify different activation orders of features in a single configuration, but would not be able to generate test suites that cover transitions *between* configurations.

The research questions explored in this vision paper are:

- RQ1 *Is CIT able to produce test suites that consistently find* behavioural transition errors *in dynamically adaptive context-aware systems?*
- RQ2 *If not, what alternative test generation approach could produce more pertinent test suites?*
- RQ3 *How feasible is this new test generation approach?*

II. Running example

Throughout this paper we demonstrate our work on the running example of a Risk Information System (RIS) inspired by Duhoux et al. [14], [15]. This system alerts users in case of emergencies and instructs them how to act in such situations. The system's core functionalities consist of *Widget*s to display safety instructions and an *Emergency Level* which depends on the emergency at hand. Fig. 1 depicts its context-feature model using a feature-based context-oriented modelling notation [2].

Before presenting some of the errors that may occur in this running example, we briefly sketch the system's intended behaviour in a bit more detail. If the user has a *GPS connection*, a *Map* widget is displayed in the application. This widget may enhance the user's experience by showing an itinerary to a safe zone, for example. Two hazards handled by our prototype are *Cold weather* with a *Low* emergency level and *Flood* with *High* emergency. *Low* emergency implies that only warnings will pop-up on a user's device, whereas a *High* emergency level triggers a more invasive behaviour such as loud sound alerts. These emergency messages warn a user about the whereabouts of the hazard, its duration or intensity. For the sake of simplicity, assume we represent the *Emergency Level* by a simple variable in our system. *No* emergency level would correspond to a value of 0, *Low* emergency to a value 1, and 2 if there is a *High* emergency (if both a *Low* and *High* emergency occur simultaneously, *High* emergency would take priority). Emergency instructions are only sent when a user is inside an *Impacted* zone. In case of multiple hazards, the different sets of specific instructions are all displayed.

For a better user experience, we want our *Widget*s to behave consistently. For example, the *Map* should always be shown above the *Instructions*, but if the *Map* is not active all *Instructions* should be at the top of the application.

Now assume that our system contains the following three behavioural errors caused by an incorrect implementation:

- T1 A behavioural transition error when deactivating feature *High* and activating feature *Low* simultaneously (defined as transition |-High, +Low|) but where the emergency level remains set to value 2 rather than dropping back to 1.
- T2 A behavioural transition error when deactivating feature *Map* and activating feature *Instructions* at the same time (defined as transition |-Map, +Instructions|) but where the widget *Instructions* remains at the bottom of the screen, leaving an empty space above it where the map used to be.
- I3 A behavioural interaction error when both features *InstructionsColdWeather* and *InstructionsFlood* are activated simultaneously in a same configuration (defined as interaction {+InstructionsColdWeather, +InstructionsFlood}), but instead of both instructions being displayed, the instructions for flood overwrite and delete those for cold weather.

Before illustrating these errors in more detail, we must define a deterministic (de)activation order, to avoid that transitions where multiple features are activated or deactivated simultaneously behave differently between two executions. This order is defined by the underlying implementation architecture. For example, for FBCOP [2] systems the (de)activation order is such that parent features are activated before their child features, and inversely upon deactivation. In a transition between two configurations (the *initial* and *final* configuration), when an activation and a deactivation occur simultaneously, we first deactivate features then activate features.

We now show pseudo-codes[1] that cause behavioural transition errors **T1** and **T2**. Comments in this pseudo-code show potential corrections for each error. In addition to its core behaviour (not shown here), each involved feature is divided into an activation and a deactivation code fragment, executed at activation and deactivation time, respectively.

Pseudo-code 1: High_deactivation

```
1  if Low is activated in the final configuration then
2  |    emergency_level = 1
3  else
4  |    emergency_level = 0
5  end
```

Pseudo-codes 1 and 2 illustrate error **T1**. Assume the initial configuration is {+High, -Low} with a (correct) emergency

[1]A more complete implementation can be found at https://github.com/PierreMartou/CoAST-TransitionTesting-2023.

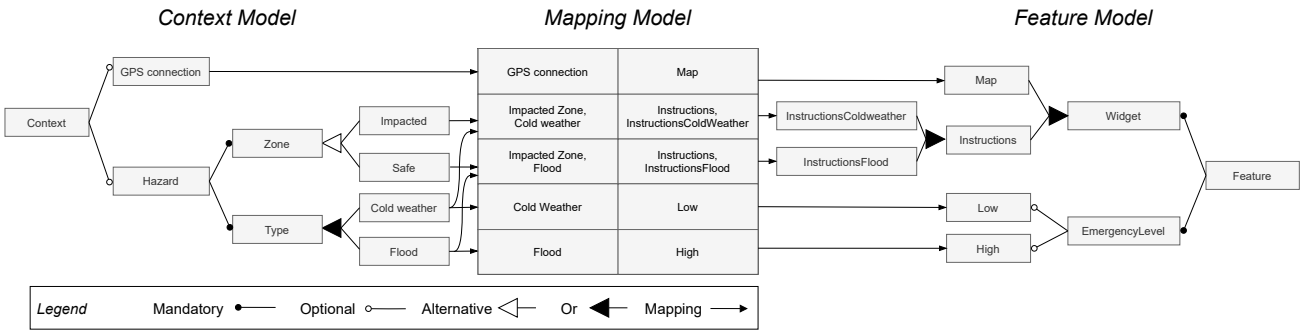

Fig. 1. Context-feature model of our Risk Information System (context model on the left, feature model on the right and mapping model in between).

Pseudo-code 2: Low_activation

```
    // if emergency_level == 0
1   if emergency_level < 2 then
2   |   emergency_level += 1
3   end
```

level of 2. After deactivating *High*, pseudo-code 1 causes the emergency level to be set to 1. Then, with the activation of *Low* (pseudo-code 2), as the emergency level is less than 2, its value gets incremented, erroneously setting the emergency level to 2 instead of 1. But note how no error can be found if this final configuration is reached through any other transitions. For example, the transition between the two configurations {+Low, +High} and {+Low, -High} (identical final configuration) will correctly set the emergency level to 1 after the execution of the deactivation of *High*. Even more interestingly, the deactivation of *High* and activation of *Low* must be simultaneous to trigger the erroneous behaviour mentioned above. The sequence of configurations {-Low, +High} (same initial configuration), followed by {+Low, +High} and {+Low, -High} (same final configuration) will behave exactly as expected, with an emergency value of 1.

Pseudo-code 3: Map_deactivation

```
1   widget_Map.delete()
2   if Instructions is activated in the initial
      configuration and in the final configuration then
3   |   widget_Instructions.delete()
4   |   widget_Instructions.spawn(above)
5   end
```

Error **T2** associated to the transition |-Map, +Instructions| is similar. Initially, only the widget *Map* is present and gets deleted on deactivation of *Map* (pseudo-code 3). Then, pseudo-code 4 adds the widget *Instructions* upon activation of *Instructions*, but this transition leads to the if-condition being entered erroneously and the widget *Instructions* being placed below, thus leaving an empty space above it, which is undesired behaviour. But when transitioning from configuration {-Map, -Instructions} to configuration {-Map, +Instructions} (same final configuration), pseudo-code 4 works correctly and the widget will be placed correctly at the top.

Observe how both errors are simple misjudgements in an

Pseudo-code 4: Instructions_activation

```
    // if Map is activated in the end
       configuration
1   if Map is activated in the initial configuration or
      in the final configuration then
2   |   widget_Instructions.spawn(below)
3   else
4   |   widget_Instructions.spawn(above)
5   end
```

if-condition, that work in most cases, and are typical errors that could occur when implementing context-aware systems.

III. Pertinence of CIT

Our first research question was: *is CIT able to produce test suites that consistently find* behavioural transition errors *in dynamically adaptive context-aware systems?* To answer **RQ1**, we illustrate how a test suite, generated by pairwise CIT, that covers all pairs of features, will not necessarily find transition errors **T1** and **T2** above. As we are mainly searching for specific transitions, Table I describes the CIT-generated test suite in terms of added and removed contexts and features between consecutive test configurations. In this particular suite of configurations, after an initially empty configuration, the system detects a *Cold weather* hazard (test 1), triggering the application to provides *Instructions* on *Cold weather* with *Low* emergency level. Later, the system detects a *Flood* (test 2), etc.

With test 4 this test suite can detect the behavioural interaction error **I3** related to the interaction {+InstructionsColdweather, +InstructionsFlood}. However, when observing the *Features* column, this test suite never covers the transitions |-High, +Low| or |-Map, +Instructions|, and therefore does *not* detect either of the two behavioural transition errors **T1** or **T2**, although it *does* cover all interactions between features.

Of course, CIT was never claimed to be perfect. To assess its efficacy to detect behavioural transition errors, we reuse a greedy algorithm which was successfully applied to FBCOP systems [7] to generate 10000 test suites. We then compute the percentages of generated test suites that contain either the transition |-High, +Low| (**T1**), the transition |-Map, +Instructions| (**T2**), or both (**T1&T2**). We also compute the transition coverage, i.e. the number of covered transitions

TABLE I
TEST SUITE THAT COVERS ALL INTERACTIONS, IN TERMS OF TRANSITIONS. *Instructions* IS SHORTENED TO *Instr*.

Test	Contexts	Features
0	*(No contexts active.)*	*(No features active.)*
1	+Zone, +Impacted, +Hazard, +Type, +Coldweather	+Widget, +Instr, +InstrColdweather, +EmergencyLevel, +Low
2	-Coldweather, +Flood	-InstrColdweather, -Low, +InstrFlood, +High
3	-Impacted, +Safe, +GPSconnection	-Instr, -InstrFlood, +Map
4	-Safe, +Impacted, +Coldweather	+Instr, +InstrFlood, +InstrColdweather, +Low
5	-Flood	-InstrFlood, -High
6	-Zone, -Impacted, -Coldweather, -Hazard, -Type	-Instr, -InstrColdweather, -Low
7	+Safe, +Zone, +Coldweather, +Hazard, +Type	+Low
8	+Flood	+High

divided by the number of all possible transitions, averaged over the 10000 test suites (*Average Coverage*).

In practice, rearrangement of configurations in a test suite is required to efficiently apply the test suite to a system. Be it for minimising reconfiguration effort [7] or finding errors as fast as possible [16], the order in which the test scenarios are applied to the system is increasingly important [17]. For this reason, table II distinguishes four different rearrangements of configurations: *Unordered* means no rearrangement, *Random* is random shuffling, *Dissimilarity* is maximising the dissimilarity between subsequent test configurations to try and find errors more quickly [18], and *Cost* is minimising the reconfiguration cost, i.e. the number of context switches between all test configurations [7]. This last rearrangement is the most relevant one, as it was developed specifically for FBCOP systems to reduce reconfiguration costs.

TABLE II
PERCENTAGE OF TEST SUITES THAT CONTAIN T1, T2, BOTH AND THEIR AVERAGE COVERAGE, WITH DIFFERENT REARRANGEMENTS.

	T1	T2	T1 & T2	Average Coverage
Unordered	74%	100%	74%	61.19%
Random	58%	84%	47.6%	48.66%
Dissimilarity	82%	100%	82%	73.63%
Cost	56%	58%	34%	22.93%

Clearly, *Random* and *Cost* rearrangements have similarly low results, while *Unordered* and *Dissimilarity* have a much better chance at finding **T1** and **T2**. This is the result of the former two having much less context switches (especially *Cost*), and hence much less transitions overall, than rearrangements aiming at diversity between consecutive test configurations.

As expected, the chances to find **T1** and **T2** are approximately the multiplication of the chances for *T1* and for *T2*. Consequently, the chance of covering all 50 valid transitions in a single test suite is nearly zero. As for the *Average Coverage*, while *Dissimilarity* achieves a moderately high coverage of 73.63%, coverage of both *Random* and *Cost* remain below 50%. This low coverage answers our **RQ1**: no, CIT is not able to produce test suites that consistently find behavioural transition errors in context-aware and dynamic systems, especially not when applying rearrangements to reduce the reconfiguration cost.

IV. COMBINATORIAL TRANSITION TESTING

We saw previously that CIT does not necessarily cover all possible behavioural transition errors. To answer **RQ2**, i.e. *what alternative test generation approach could produce more pertinent test suites?*, we present a new approach called *Combinatorial Transition Testing* (CTT) that aims at covering all possible transitions. We define it by highlighting its differences with CIT. In CIT, an interaction tuple $\{v_1, v_2, ..., v_n\}$ of size *n* contains *n* value assignments to features.

Definition 1: A *value assignment* v_i is a pair *(value, feature)* with *value* being either activated ("+") or deactivated ("-"). Its opposite assignment $-v_i$ is the pair *(-value, feature)* where *-value* is the opposite activation status.

A *valid* interaction tuple is then an assignment of values that satisfy all constraints (i.e., a valid configuration that includes this tuple exists). A tuple is *covered* in a configuration if its assignment of values is totally included in this configuration. For example, in a system with the features *A*, *B* and *C* and no constraints, the interaction {+A, -B} is covered by the configuration {**+A, -B**, +C}.

Similarly, n-wise *Combinatorial Transition Testing* should generate a test suite that covers all *transitions* of size n. The main difference lies in a new definition of coverage:

Definition 2: A transition $|v_1, v_2, ..., v_n|$ is covered in a test suite if, in two consecutive configurations, the second configuration includes the assignment of values $\{v_1, v_2, ..., v_n\}$ while the first configuration includes the opposite assignment of values $\{-v_1, -v_2, ..., -v_n\}$.

For example, the two consecutive configurations {**-A, +B**, +C} and {**+A, -B**, +C} will cover the transition |+A, -B|. Note that in this example, the first configuration {**-A, +B**, +C} must be valid for the transition to be possible. More generally:

Definition 3: A transition pair is valid if and only if the specified assignment of values and its opposite satisfy all system constraints.

Finally, let us examine the relation between transition and interaction testing. Observe that if the transition |+A, -B| is covered, then the *corresponding* interaction {+A, -B} is necessarily covered. It leads to our final definition:

Definition 4: All transitions $|v_1, v_2, ..., v_n|$ have corresponding interactions $\{v_1, v_2, ..., v_n\}$ (invertible interactions). A test suite that covers a transition necessarily covers the corresponding interaction. Interaction tuples whose opposite is an invalid interaction have no corresponding transition (non-invertible interactions). Interactions are either invertible or not.

V. FEASIBILITY

Our final **RQ3** is *how feasible this new test generation approach is?* To answer it, we discuss three challenges and build the requirements of an *ideal* test generation approach.

Firstly, from Definition 4 we learn that a test suite that covers all possible transitions does not necessarily cover all

interactions, just like CIT does not cover all transitions. Hence, a first desirable property of an ideal test generation approach is coverage of all transitions and all interactions.

Let us examine the relation between the number of transitions ($\#transitions_n$) and interactions ($\#interactions_n$) of size *n*, and the number of invertible ($\#invert\_int_n$) and non-invertible interactions ($\#non\_inv\_int_n$):

$$\#invert\_int_n = \#transitions_n \quad (1)$$

$$\#invert\_int_n + \#non\_inv\_int_n = \#interactions_n \quad (2)$$

From these equation 1 and 2 we can infer Equation 3:

$$\#transitions_n + \#non\_inv\_int_n = \#interactions_n \quad (3)$$

Equation 3 shows that an ideal generation algorithm that covers both transitions (i.e. invertible interactions) and non-invertible interactions aims at covering exactly the same number of tuples as typical CIT algorithms. The scale of coverage is the same as with CIT, which was already shown to be achievable to a high degree through different techniques [19].

A second challenge arises when examining n-wise CTT. While n-wise CIT necessarily always covers all pairs of size inferior to n, this is not the case of n-wise CTT. To illustrate this, assume a feature model with two features *A* (optional) and *B* (mandatory). In CIT, the valid tuples of size *2* are {+A, +B} and {-A, +B}, both non-invertible interactions as feature *B* cannot be deactivated. The only valid transitions are |+A| and |-A|, of size 1, which would not be included in standard 2-wise CTT. This can be generalised to any group of $< n$ features whose status can only be switched when the rest of the system is fixed to a specific activation state.

Although in practice this case would be quite uncommon, to ensure completeness of our approach an ideal algorithm of *n*-wise CTT should include all valid tuples of size $\leq n$ ($\#transitions_{\leq n}$). Contrary to initial intuition, this enlarged coverage has in fact an identical scaling as before, as shown by equations 4 to 6. Indeed, the sum of a geometrical suite scales only with its last term (Equation 6): the number of transitions of size $< n$ is negligible compared to all transitions of size n.

$$\#transitions_n \sim \mathcal{O}(\#features^n) \quad (4)$$

$$\#transitions_{\leq n} \sim \mathcal{O}(\sum_{i=1}^{n} \#features^i) \quad (5)$$

$$\mathcal{O}(\sum_{i=1}^{n} \#features^i) \simeq \mathcal{O}(\#features^n) \quad (6)$$

A third challenge concerns rearrangement. As mentioned in Section III, prioritisation is increasingly important in CIT. In CTT however, we can assume that reordering a test suite *after* generation is impossible, as each consecutive test configuration would be built to cover specific transitions and changing their order would result in a different, surely smaller, coverage. Table II even showed that different rearrangements yield vastly different results in coverage. Hence, prioritisation ought to be directly integrated into the test suite generation process, rather than a posteriori. To achieve this, promising directions include weighted sampling approaches [19] or multi-objective algorithms (e.g. genetic algorithms [20]).

To conclude, this vision paper argued the need for Combinatorial Transition Testing to complement Combinatorial Interaction Testing by showing how transition errors can remain undetected in dynamic and context-aware systems. We sketched the outline of an algorithm that covers both interactions and transitions of size $\leq$ n, and confirmed its feasibility. This articles paves the way for future research on CTT, and the next step will be the implementation and validation of such an algorithm on real cases.

REFERENCES

[1] G. Salvaneschi, C. Ghezzi, and M. Pradella, "Context-oriented programming: A software engineering perspective," *JSS*, vol. 85, no. 8, 2012.

[2] B. Duhoux, "Feature-based context-oriented software development," Ph.D. dissertation, 2022.

[3] 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.

[4] H. Hartmann and T. Trew, "Using feature diagrams with context variability to model multiple product lines for software supply chains," in *SPLC '08*. IEEE, 2008.

[5] R. Capilla, O. Ortiz, and M. Hinchey, "Context variability for context-aware systems," *Computer*, vol. 47, no. 2, pp. 85–87, 2014.

[6] K. Mens, R. Capilla, H. Hartmann, and T. Kropf, "Modeling and managing context-aware systems' variability," *IEEE Software*, vol. 34, no. 6, pp. 58–63, 2017.

[7] P. Martou, K. Mens, B. Duhoux, and A. Legay, "Test scenario generation for feature-based context-oriented software systems," *JSS*, p. 111570, 2022.

[8] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys (CSUR)*, vol. 43, no. 2, pp. 1–29, 2011.

[9] 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.

[10] I. de Sousa Santos, R. M. de Castro Andrade, L. S. Rocha, S. Matalonga, K. M. de Oliveira, and G. H. Travassos, "Test case design for context-aware applications: Are we there yet?" *Information and Software Technology*, vol. 88, pp. 1–16, 2017.

[11] R. Seiger and T. Schlegel, "Test modeling for context-aware ubiquitous applications with feature petri nets," in *MODIQUITOUS*. Citeseer, 2012.

[12] D. R. Kuhn, J. M. Higdon, J. F. Lawrence, R. N. Kacker, and Y. Lei, "Combinatorial methods for event sequence testing," in *ICST '12*. IEEE, 2012, pp. 601–609.

[13] A. Elyasaf, E. Farchi, O. Margalit, G. Weiss, and Y. Weiss, "Combinatorial sequence testing using behavioral programming and generalized coverage criteria," *arXiv preprint arXiv:2201.00522*, 2022.

[14] B. Duhoux, K. Mens, and B. Dumas, "Feature visualiser: An inspection tool for context-oriented programmers," in *COP '18*. New York, NY, USA: Association for Computing Machinery, 2018, p. 15–22.

[15] B. Duhoux, K. Mens, B. Dumas, and H. S. Leung, "A context and feature visualisation tool for a feature-based context-oriented programming language," in *SATTOSE '19*, ser. CEUR Workshop Proceedings, vol. 2510. CEUR-WS.org, 2019.

[16] C. Catal and D. Mishra, "Test case prioritization: a systematic mapping study," *Software Quality Journal*, vol. 21, no. 3, pp. 445–478, 2013.

[17] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *STVR*, vol. 22, no. 2, pp. 67–120, 2012.

[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.

[19] E. Baranov, A. Legay, and K. S. Meel, "Baital: an adaptive weighted sampling approach for improved t-wise coverage," in *ESEC/FSE 2020*, 2020, pp. 1114–1126.

[20] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon, "Multi-objective test generation for software product lines," in *SPLC '13*, 2013, pp. 62–71.