A Variability Perspective of Mutation Analysis

Xavier Devroey, Gilles Perrouin, Maxime Cordy, Mike Papadakis, Axel Legay, Pierre-Yves Schobbens

A Variability Perspective of Mutation Analysis Technical Report

Xavier Devroey
PReCISE Research Center
University of Namur, Belgium
xde@info.fundp.ac.be

Gilles Perrouin*
PReCISE Research Center
University of Namur, Belgium
gpe@info.fundp.ac.be

Maxime Cordy†
PReCISE Research Center
University of Namur, Belgium
mcr@info.fundp.ac.be

Mike Papadakis
SnT, SERVAL Team
University of Luxembourg
mike.papadakis@uni.lu

Axel Legay
INRIA Rennes, France
axel.legay@inria.fr

Pierre-Yves Schobbens
PReCISE Research Center,
University of Namur, Belgium
pys@info.fundp.ac.be

ABSTRACT

Mutation testing is an effective technique for either improving or generating fault-finding test suites. It creates defective or incorrect program artifacts of the program under test and evaluates the ability of test suites to reveal them. Despite being effective, mutation is costly since it requires assessing the test cases with a large number of defective artifacts. Even worse, some of these artifacts are behaviourally "equivalent" to the original one and hence, they unnecessarily increase the testing effort. We adopt a variability perspective on mutation analysis. We model a defective artifact as a transition system with a specific feature selected and consider it as a member of a mutant family. The mutant family is encoded as a Featured Transition System, a compact formalism initially dedicated to model-checking of software product lines. We show how to evaluate a test suite against the set of all candidate defects by using mutant families. We can evaluate all the considered defects at the same time and isolate some equivalent mutants. We can also assist the test generation process and efficiently consider higher-order mutants.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*

Keywords

Mutation Testing, Featured Transition Systems

1. BEHAVIOURAL MUTATION TESTING

Traditionally, research on mutation analysis focuses on testing implementations in a given programming language. However, the recent years this situation has changed [28] and mutation has also been applied for testing various design models. This situation is reported in the recent survey of Jia and Harman [20] motivating the need for additional research on applying mutation testing on program artifacts other than code. Among the several works mentioned in the mutation testing survey [20], Fabbri et al. [15] proposed 9 mutation operators injecting faults in states, events and outputs of Finite State Machines (FSMs). Other formalisms were targeted such as EFSMs [6], statecharts [16] or UML state machines [2].

The formal nature of these specification languages was exploited to enable the use of verification tools in the context of mutation testing. For example, Amman et al. demonstrated the use of the model-checker SMV by mutating specifications and using counterexamples (violation of a temporal logic property) as test cases [5]. In addition, simulation techniques exists to detect and remove equivalent mutants. Aichernig et al. improved such simulation techniques using the ioco conformance relation [3].

*FNRS Postdoctoral Researcher
†FNRS Research Fellow

FSE'14 , November 16–22, 2014, Hong Kong, China

2. FEATURED TRANSITION SYSTEMS

Software Product Line (SPL) engineering is a sub-discipline of software engineering based on the idea that products of the same family can be built by systematically reusing assets, some of them being common to all members whereas others are only shared by a subset of the family. Such variability is commonly captured by the notion of *feature*. Individual features can be specified using languages such as UML, while their inter-relationships are organized in a Feature Diagram (FD) [22].

The main challenge in SPL engineering is to deal with the combinatorial explosion induced by the number of possible products (2^N for N features in worst case). To face this problem, Classen et al. introduced Featured Transition Systems (FTSs) [8] in order to perform efficient behavioural model checking for SPLs. FTSs are Transition Systems (TSs) where each transition is labelled with a feature expression specifying which products of the SPL may fire the transition. A FTS is thus a compact representation of the behaviour of a whole product line. Formally, an FTS is a tuple $(S, Act, trans, i, d, \gamma)$, where S is a set of states; Act a set of actions; $trans \subseteq S \times Act \times S$ is the transition relation (with $(s_1, \alpha, s_2) \in trans$ sometimes noted $s_1 \xrightarrow{\alpha} s_2$); d is

a FD; and $\gamma : trans \rightarrow [\![d]\!] \rightarrow \{\top, \bot\}$ is a total function labelling each transition with a boolean expression over the features, which specifies the products that can execute the transition ($[\![d]\!]$ corresponds to the semantics of the FD d, i.e., all the different products that may be derived from d). Regarding the initial state, we define a total function $init : S \mapsto ([\![d]\!] \mapsto \top, \bot)$ that indicates if a state $i \in S$ is an initial state for a product $p \in [\![d]\!]$. This function allows to model mutants that change the initial state of the system. A model checker for FTSs has been implemented in ProVeLines [10], a product line of model checkers for the verification of SPLs' behavioural models.

3. APPROACH

The key aspect of our vision is to ***consider mutants as member of a family rather than perceived and analysed in isolation***. Thus, we envision to ***exploit and adapt variability-management and analysis techniques to perform better and faster mutation testing***, be it for single systems or highly-configurable systems such as software product lines.

Our vision can be seen as a generalisation at the model level of mutant schemata proposed by Untch et al [30], but with additional benefits such as programming language independence, higher-order mutation, weak mutation automated test-case generation and equivalent mutants analysis. We will see these assets in the next section, while we focus here on modelling mutants families.

Let us consider a simple drink vending machine, whose behaviour is represented by the original transition system on left of Figure 1. We consider the application of three transition systems mutation operators [15], removing a state, (see `a) StateMissing`), changing an event by another one (see `b) EventExchanged`) and changing the initial state of the system (see `c) WrongStartState`). These three mutants together with the original transition system can be compactly modelled using the FTS formalism presented in the previous section as depicted on the right of Figure 1. Our mutant family is therefore composed of a feature model and its accompanying FTS. As opposed to "usual" product lines, features do not map to a particular option or functionality to be be offered by the system but rather a mutation that can be applied to it.

We organize our feature hierarchy in two (the "root" layer, compulsory, is not really significant) layers: the "mutation operator" level where we define mutation operators available for the mutation family (features: `sm`, `eex`, `wass`) and the "mutation instance" where each leaf feature (e.g. `sm_4`) represents a specific mutation to be applied to the model (e.g. `sm_4` stands for state 4 is missing). It is of course possible to use more sophisticated variability management constructs to exclude/require mutants or force the selection of only one mutant instance (using group cardinalities) for each mutation operator.

Once mutations are described in terms of a feature diagram, we can model the impact of mutations on the behaviour of the systems, by building a mutation FTS. The first step is to "lift" our original transition system into an equivalent FTS. To do so, we tag all transitions with the root feature whose only value is `true` since it is mandatory. Then for each mutation concerned we either tag the concerned transitions with mutations instances features or their negation (this how we mapped the missing state 4 corresponding to mutant a)) or add a new transition (`exx_t_o`) to describe an alternative behaviour when this mutant is applied.[1]

This modelling proposal allows to finely configure the desired mutants by activating or deactivating features in the feature model. Exploiting our research on configurators' generation [7], we can derive tailored mutants configurators. These configurators would allow testers to tune mutant generation exactly the same way they would configure their future car, while avoiding configurations forbidden by the feature model. Our approach can be used to describe first-order mutants (only one feature is selected at a time) or higher-order mutants (several features can be selected). Several mutations can also be modelled on the same element (e.g. changing both the destination and the event of a transition), by detailing feature expressions on transitions.

4. APPLICATIONS & CHALLENGES

4.1 Scalable Mutation Analysis

Variability-aware Mutation Analysis. The computational cost of executing a large number of mutants has been recognised as an obstacle to the practical application of mutation testing. Handling a large number of products is also challenging for SPL testing and was the *raison d'être* of FTS. Evaluating all products at the same time allowed significant gains (from 2 to 1000 times faster) for model-checking purposes [8]. Recent work in SPL testing [23, 24, 27] confirms the potential of this research direction. While there are similarities with Mutant Schemata [30] at the program level, runtime execution differs. Indeed, in mutant schemata's approach mutants are independently run against each test case while as we traverse FTS transitions feature expressions indicate which mutants are being killed and which ones are surviving. Thus, we do not only compactly represent all the mutants as for metaprograms but also save time during execution by exploiting commonalities between mutants' behaviour. Jia and Harman claim that bottlenecks moved from compilation time to runtime [19]. Our vision clearly adheres to this claim.

Selective Mutation. As surveyed by Jia and Harman [20], selective mutation approaches are popular amongst cost reduction techniques. Selective mutation aims at reducing the number of mutants to consider using various criteria (e.g. random sampling, clustering or limiting the number of applicable mutation operators). Such selections can be easily modelled and verified (thanks to their translation in constraint solvers' inputs) using feature diagrams. For example, it can be interesting to assess criteria commonly applied to feature models such as *t-wise coverage* [29, 9] derived from combinatorial interaction testing. T-wise coverage select configurations so that all t-combinations of features are present, observing that most bugs are due to (undesired) feature interactions.

In our context, this corresponds to mutant interactions: 1-wise selection would ensure that every mutation is covered at least once, that is assessing a test suite against all first-order mutants, the classical "pairwise" (2-wise) would ensure that all possible combination of 2^{nd}-order mutants are covered. Usual t-wise approaches would not prevent higher-order (t=3,4,...) mutants from being generated. The main advantage of such techniques is that they drastically

[1]See `https://staff.info.unamur.be/xde/fts-testing/mutation.html` for more information.

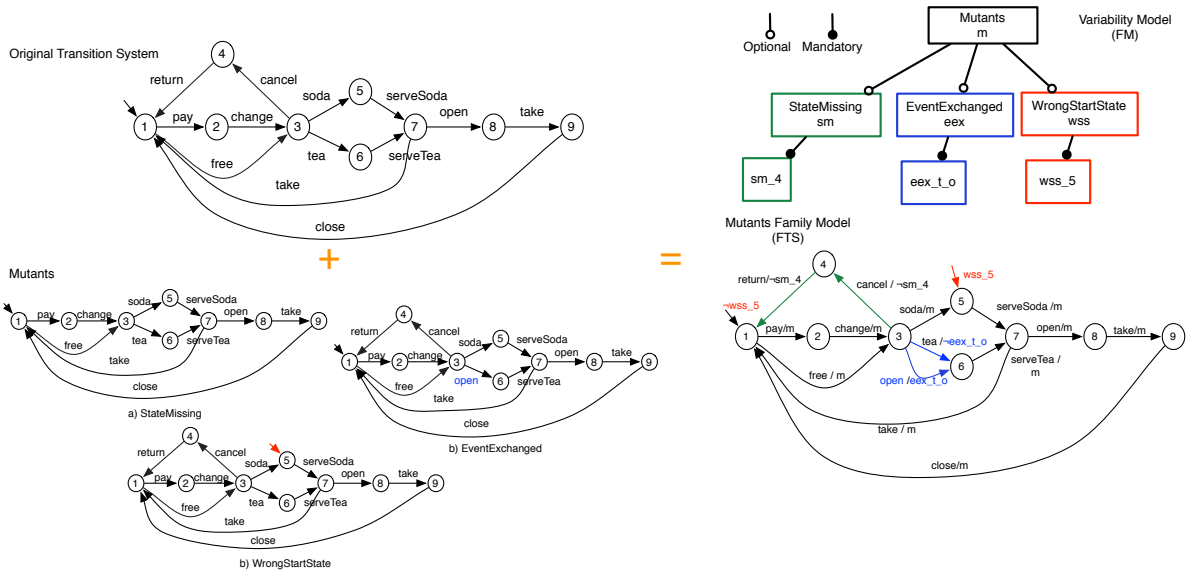

Figure 1: Approach Overview

reduce the number of mutants to consider (few hundred even for feature models allowing billions of possibilities). It is also possible to mix criteria for a more fine grained selection and to prioritize mutants selection of interest, by assigning weights [21] on mutation operators/instances in the FM and/or considering multi-objective selection [18, 20]. Mutant selection is enacted in the FTS using the *projection operator* [8] that prunes unselected transitions and related states.
Coverage criteria can also be specified on the mutant family model ensuring that some states/transitions/actions are covered. Only these will be analysed for mutation (weak mutation). This can be done either by adapting transition systems' coverage criteria [14] or formalising this search as a temporal logic formula to be checked by, e.g. our ProVeLines family of model-checkers [10].

4.2 Test Case Generation

Thanks to the formal foundations of FTSs, various test and verification techniques can be invented or re-adapted to derive efficient mutant killing test suites. In our framework, a test case is simply a trace allowed by the FTS, that is a succession of actions in which the conjunction of feature expressions accumulated over the associated transitions is satisfiable w.r.t the FD [14]. The *necessity constraint* [12], expressing difference in behaviours between the original and mutated programs, can be easily translated as an LTL formula expressing a certain condition that is satisfied on the original system (all paths in which all transitions are only labelled by `m`). Thus, all counter-examples produced by ProVeLines are behaviours killing mutants and for each behaviour, the feature expression indicating the combination of mutants covered is also provided. For weak mutation, we can negate a formula involving the particular state or action this mutant affects: counter-examples will necessarily reach the targeted mutant. This offers a complementary set of techniques to coverage and constraint-based generation [12].

4.3 Equivalent Mutants Detection

Madeyski et al. recently surveyed the Equivalent Mutant Problem (EMP) [26] and categorized existing techniques falling in three categories: detection, suggestion and avoidance. In our context, EMP is: are there products of the FD whose associated behaviour are identical? Simulation [11] and language equivalence relations may be part of the solution. Such relations can be used to detect equivalent mutants and to remove them when incrementally building the FTS as explained in Section 3. Yet, computing these relations is expensive (EXPTIME for simulations). An alternative strategy is the selection of higher-order mutants as proposed in the survey [26] and discussed above.

4.4 Challenges

Model v.s. code-based. Our vision deliberately resides at the model level as opposed to code-based mutation analysis. This allows to reason on test suites qualities early in the process and to produce platform-independent test cases. As for any model-based approach, the availability of an accurate model is crucial. We rely on model inference techniques such as n-grams derived from logs [13], automata learning approaches [25] and feature modelling reverse-engineering [1]. Since these methods are not exact, the toughest challenge is to characterize and reduce the impact of extraction issues in the testing process [17].

Mutation Analysis. While our variability-based approach can exploit all innovations in VIS selection of configurations and behaviours, the theoretical underpinnings and practical relevance of (higher-order) mutants interaction need to be researched [4]: What are their *subsuming* [20] abilities? Are they harder to kill than HOMs constructed with another techniques?

5. CONCLUSION

In this paper, we presented our vision for improving model-based behavioural mutation testing by considering mutants

as part of a family rather than individually. This approach can speed up mutation analysis by running test cases against all mutants at the same time, rather than independently, thus efficiently addressing "explosion" of the number of mutants. The proposed approach also provides additional benefits, such as the precise modelling of mutants dependencies and easy mutant selection. Thanks to the formal nature of FTS and dedicated model-checking facilities, we target an universal framework where several new and existing mutant analysis techniques can be formalized and implemented.

6. REFERENCES

[1] M. Acher, A. Cleve, G. Perrouin, P. Heymans, C. Vanbeneden, P. Collet, and P. Lahire. On extracting feature models from product descriptions. In *VaMoS*, pages 45–54, New York, NY, USA, 2012. ACM.

[2] B. K. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, R. Schlick, and S. Tiran. Killing strategies for model-based mutation testing. *STVR*, 2014.

[3] B. K. Aichernig, M. Weiglhofer, and F. Wotawa. Improving fault-based conformance testing. *Electr. Notes Theor. Comput. Sci.*, 220(1):63–77, 2008.

[4] P. Ammann, M. E. Delamaro, and J. Offutt. Establishing theoretical minimal sets of mutants. In *ICST*, pages 21–30. IEEE, 2014.

[5] P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *FEM*, pages 46–54. IEEE, 1998.

[6] N. Bombieri, F. Fummi, and G. Pravadelli. A mutation model for the SystemC TLM 2.0 communication interfaces. In *DATE*, pages 396–401. ACM, 2008.

[7] Q. Boucher, G. Perrouin, and P. Heymans. Deriving configuration interfaces from feature models: A vision paper. In *VaMoS*, pages 37–44, New York, NY, USA, 2012. ACM.

[8] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to ltl model checking. *IEEE TSE*, 39(8):1069–1089, 2013.

[9] M. Cohen, M. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *ISSTA*, pages 129–139, 2007.

[10] M. Cordy, A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Provelines: A product-line of verifiers for software product lines. In *SPLC '13 Workshops*, pages 141–146. ACM, 2013.

[11] M. Cordy, A. Classen, G. Perrouin, P.-Y. Schobbens, P. Heymans, and A. Legay. Simulation-based abstractions for software product-line model checking. In M. Glinz, G. C. Murphy, and M. Pezzè, editors, *ICSE*, pages 672–682. IEEE, 2012.

[12] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE TSE*, 17(9):900–910, 1991.

[13] X. Devroey, G. Perrouin, M. Cordy, P.-Y. Schobbens, A. Legay, and P. Heymans. Towards statistical prioritization for software product lines testing. In *VaMoS*, pages 10:1–10:7. ACM, 2014.

[14] X. Devroey, G. Perrouin, A. Legay, M. Cordy, P.-y. Schobbens, and P. Heymans. Coverage Criteria for Behavioural Testing of Software Product Lines. In *ISoLA*. Springer, 2014.

[15] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, and M. E. Delamaro. Proteum/fsm: A tool to support finite state machine validation based on mutation testing. In *SCCC*, pages 96–104. IEEE Computer Society, 1999.

[16] S. C. P. F. Fabbri, J. C. Maldonado, T. Sugeta, and P. C. Masiero. Mutation testing applied to validate specifications based on Statecharts. In *SRE*, pages 210–219. IEEE, 1999.

[17] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. Le Traon. Towards automated testing and fixing of re-engineered feature models. In *ICSE*, pages 1245–1248. IEEE, May 2013.

[18] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon. Multi-objective test generation for software product lines. In *SPLC*, pages 62–71. ACM, 2013.

[19] Y. Jia and M. Harman. MILU: A Customizable, Runtime-Optimized Higher-Order Mutation Testing Tool for the Full C Language. In *TAIC-PART*, pages 94–98, Washington, DC, USA, 2008. IEEE Computer Society.

[20] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE TSE*, 37(5):649–678, 2011.

[21] M. F. Johansen, Ø. Haugen, F. Fleurey, A. G. Eldegard, and T. Syversen. Generating better partial covering arrays by modeling weights on sub-product lines. In *MoDELS*, pages 269–284. Springer, 2012.

[22] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. Spencer Peterson. Feature-Oriented domain analysis (FODA) feasibility study. Technical report, Soft. Eng. Inst., Carnegie Mellon Univ., 1990.

[23] C. Kästner, A. von Rhein, S. Erdweg, J. Pusch, S. Apel, T. Rendel, and K. Ostermann. Toward variability-aware testing. In *FOSD*, pages 1–8. ACM, 2012.

[24] C. H. P. Kim, S. Khurshid, and D. S. Batory. Shared execution for efficiently testing product lines. In *ISSRE*, pages 221–230. IEEE, 2012.

[25] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *ICSE*, pages 501–510, New York, NY, USA, 2008. ACM.

[26] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Jozala. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *IEEE TSE*, 40(1):23–42, Jan 2014.

[27] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Exploring variability-aware execution for testing plugin-based web applications. In *ICSE*. IEEE, 2014.

[28] J. Offutt. A mutation carol: Past, present and future. *Information & Software Technology*, 53(10):1098–1107, 2011.

[29] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. L. Traon. Pairwise testing for software product lines: comparison of two approaches. *Soft. Qual. Journal*, 20(3-4):605–643, 2012.

[30] R. H. Untch, A. J. Offutt, and M. J. Harrold. Mutation analysis using mutant schemata. In *ISSTA*, pages 139–148. ACM, 1993.