# Poster: VIBeS, Transition System Mutation Made Easy

Xavier Devroey, Gilles Perrouin, Pierre-Yves Schobbens, Patrick Heymans
PReCISE Research Center, Faculty of Computer Science, University of Namur, Belgium
{xavier.devroey, gilles.perrouin, pierre-yves.schobbens, patrick.heymans}@unamur.be

***Abstract*—Mutation testing is an established technique used to evaluate the quality of a set of test cases. As model-based testing took momentum, mutation techniques were lifted to the model level. However, as for code mutation analysis, assessing test cases on a large set of mutants can be costly. In this paper, we introduce the Variability-Intensive Behavioural teSting (VIBeS) framework. Relying on Featured Transition Systems (FTSs), we represent all possible mutants in a single model constrained by a feature model for mutant (in)activation. This allow to assess all mutants in a single test case execution. We present VIBeS implementation steps and the DSL we defined to ease model-based mutation analysis.**

## I. Background

**Mutation Testing.** Mutation testing is a fault injection technique used to assess the quality of a set of test cases for a System Under Test (SUT) by seeing how many mutants of this SUT are detected (killed) by the test cases [1]. A mutant is generated from a SUT by applying a mutation operator on its source code (e.g., changing one '*' by a '/'). The mutation testing community adapted mutation analysis to the model level [2]. For instance, Fabri et al. [3] define a set of mutations operators for Finite State Machines. One of the main challenges in mutation testing is the execution of a set of test cases on a large number of mutants. Indeed, each test case will have to be executed on each mutant.

**Software Product Line (SPL).** SPL engineering is concerned by the management of variability-intensive systems. Such systems have shared and product specific assets which are regrouped in features and organized in a Feature Diagram (FD) [4]. To compactly represent the behaviour of a SPL, Classen et al. developed Featured Transition Systems (FTSs) [5]. A FTS is a Transition System (TS), where each transition is tagged with a feature expression (represented as a boolean expression over features of the system) specifying which products may or may not fire the transition.

## II. Mutants & FTS

In our previous work [6], we suggested to represent mutants using the FTS formalism, by tagging transitions with feature expressions specifying which mutant(s) may fire the transition or not. The goal is to save execution time by exploring common behaviour amongst mutants only once. Fig. 1 illustrate our featured mutant modelling approach: we consider 3 mutation operators: *StateMissing* which removes a state from the base model; *ActionExchange* which replaces an action on a transition in the base model by another action;

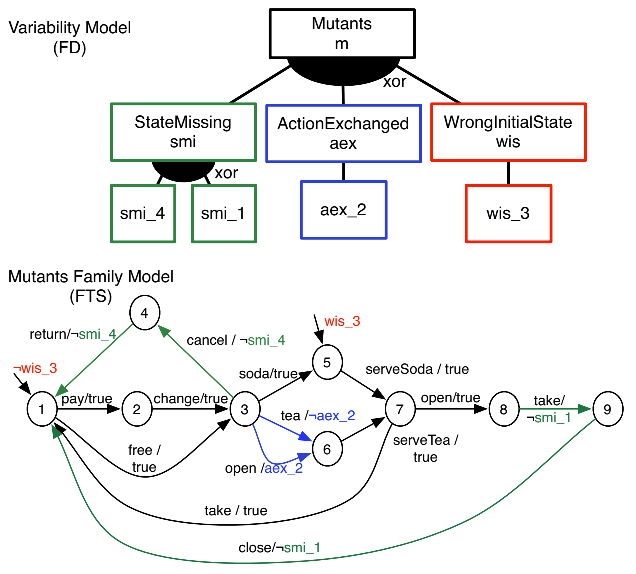

Fig. 1. Mutants Modelling with FTS [6]

and *WrongInitialState* which modifies the initial state of the base model. The *StateMissing* operator has been applied 2 times and the 2 others only once. The FD on top of Fig. 1 presents the different *applications* of the operators on the base model (grouped by operator for readability): *smi_4*, *sm_5*, *aex_2*, and *wis_5*. The FTS in Fig. 1 presents the result of this application to the base model: in place of directly modifying the base model (by removing a state, changing an action, and changing the initial state), the operators have modified feature expressions and transitions in the mutants FTS in such a way that only if the feature corresponding to the *application* of an operator on the base model is selected in the FD, the mutation is active in the FTS. For instance, transition $(s_3 \xrightarrow{cancel/\neg smi\_4} s_4)$ may not be fired (making $s_4$ unreachable) if feature *smi_4* is selected.

In model-based testing, the set of test cases is first defined at an abstract level on the model of the SUT and made concrete afterwards to match input values of the implementation [1]. To assess the quality of this set, one may perform an abstract execution of each test case on the FTS. Each execution will produce a feature expression representing all the mutants that are not killed by the test case. For instance, considering the test set $ts = \{tc_1 = (pay, change, cancel, return); tc_2 = (free, tea, serveTea, take)\}$: the execution of $tc_1$ on the FTS in Fig. 1 will produce the feature expression $\neg smi_4$, stating that mutants satisfying this feature expression are still alive. In the same way, executing $tc_2$ will produce the feature expression $\neg aex_2$. Mutants satisfying one of the feature expressions generated when executing test cases on the mutants FTS are not killed by those test cases [7]. To get the list of the mutants still alive, one will have to conjunct the mutants FD and the disjunction of the generated feature expressions, e.g., $d_{mutants} \wedge (\neg smi_4 \vee \neg aex_2)$, where $d_{mutants}$ corresponds to the equivalent feature expression of the FD in Fig. 1.

## III. Implementation in VIBeS

The different mutation operators [7] are implemented in our Variability-Intensive Behavioural teSting (VIBeS) framework (in Java) [8]. The principle is the same for each operator:

1) the operator is created using a Transition System (TS) and one or more *selection strategies* according to the type of operator (e.g., *StateMissing* operator will need a state selection strategy to select the state that will be removed);
2) the user has to call the `apply()` method, this will perform the mutation by selecting elements (transitions, actions, and/or states) and generate a unique key for this mutation (used in the feature name in the mutants FTS);
3) the method `result()` returns a fresh TS, representing the result of the mutation;
4) the method `transpose(FTS)` alters the given mutants FTS (representing the existing mutants) to add the current mutation returned by `result()` by adding transitions and or modifying feature expressions on existing transitions (transformations for each operator may be found in [7]).

To ease the use of the VIBeS API, we defined a small Java DSL to create and manipulate TSs and perform mutation on them. For example, the following code loads a TS model from an XML file and applies an ActionExchange mutation operator on it:

```
TransitionSystem ts = loadTransitionSystem("
    model.ts");
MutationOperator op = actionExchange(ts)
        .transitionSelectionStrategy(RANDOM)
        .actionSelectionStrategy(RANDOM)
        .done();
```

The user may perform the mutation using this operator and/or perform an update of the mutants FTS, equal to the original TS before the first mutation:

```
FTS mutantFts = new FTS(ts);
op.apply();
// Return the mutant TS (fresh TS)
TransitionSystem mutant = op.result();
// Update the Mutants FTS
mutantFts = op.transpose(mutantFts);
// Get the feature name of the mutation
String featName = op.getFeatureId();
```

It is also possible to use a configuration file to specify mutants generation (operators, selection strategies, and number of mutants)[1]:

```
configure("config.xml)
        // mutants fts output file (optional)
        .ftsMutant("mutants.fts")
        // mutants FD output file (optional)
        .tvlMutant("mutants.tvl")
        // TS mutants output dir. (optional)
        .outputDir("./ts-mutants/")
        // the TS to mutate
        .mutate(ts);
```

To get all the mutants alive after executing a test case, the programmer may call:

```
FExpression alive = getAliveMutants(testCase,
    mutantsFts, origInitStateName);
```

The original initial state name is needed because of the *WrongInitialState* mutation operator which may have change the initial state in the mutants FTS. The `alive` feature expression may then be used as a constraint for the mutants FD to get the number of mutants alive.

## IV. Conclusion

In this paper we introduced VIBeS, a variability-based framework to ease mutation analysis transition systems. We exhibited the first steps of its implementation as well as a JAVA DSL easing mutant modelling and analysis. Along with the validation of our ideas on actual models, there are two items on the future work agenda. First, we would like to study higher-order mutation [2]. This can be easily modelled in the FD by replacing *xor* relations by cardinalities ($[x, y]$). Yet, challenges concern scalable higher-order mutant equivalence and selection of stubborn mutants difficult to kill. The second item concerns the application of model checking techniques using ProVeLines [5], to generate mutant-killing test cases as counterexamples violating a given temporal property.

## References

[1] A. P. Mathur, *Foundations of software testing*. Pearson Education, 2008.

[2] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE TSE*, vol. 37, no. 5, pp. 649–678, 2011.

[3] S. Fabbri, J. C. Maldonado, and M. E. Delamaro, "Proteum/FSM: a tool to support finite state machine validation based on mutation testing," in *SCCC '99*, 1999, pp. 96–104.

[4] K. Kang, S. Cohen, J. Hess, W. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Carnegie-Mellon Univ., Soft. Eng. Inst., Tech. Rep., 1990.

[5] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. J.-F. Raskin, "Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking," *IEEE TSE*, vol. 39, no. 8, pp. 1069–1089, 2013.

[6] X. Devroey, G. Perrouin, M. Cordy, M. Papadakis, A. Legay, and P.-Y. Schobbens, "A Variability Perspective of Mutation Analysis," in *FSE'14*. ACM, 2014, pp. 841–844.

[7] X. Devroey and G. Perrouin, "Mutation Testing Using Featured Transition Systems," PReCISE, University of Namur, Namur, Belgium, Tech. Rep., 2014. [Online]. Available: https://projects.info.unamur.be/vibes/mutation.html

[8] "VIBeS: Variability Intensive Behavioural teSting," jan 2015. [Online]. Available: https://projects.info.unamur.be/vibes/

[1]An example of mutation configuration file may be downloaded at https://projects.info.unamur.be/vibes/mutation.html