# A Benchmark for Active Learning of Variability-Intensive Systems

Shaghayegh Tavassoli
sh.tavassoli@ut.ac.ir
University of Tehran
Tehran, IR

Carlos Diego N. Damasceno
d.damasceno@cs.ru.nl
Radboud University Nijmegen
Nijmegen, NL

Mohammad Reza Mousavi
mohammad.mousavi@kcl.ac.uk
King's College London
London, UK

Ramtin Khosravi
r.khosravi@ut.ac.ir
University of Tehran
Tehran, IR

## ABSTRACT

Behavioral models are the key enablers for behavioral analysis of Software Product Lines (SPL), including testing and model checking. Active model learning comes to the rescue when family behavioral models are non-existent or outdated. A key challenge on active model learning is to detect commonalities and variability efficiently and combine them into concise family models. Benchmarks and their associated metrics will play a key role in shaping the research agenda in this promising field and provide an effective means for comparing and identifying relative strengths and weaknesses in the forthcoming techniques. In this challenge, we seek benchmarks to evaluate the efficiency (e.g., learning time and memory footprint) and effectiveness (e.g., conciseness and accuracy of family models) of active model learning methods in the software product line context. These benchmark sets must contain the structural and behavioral variability models of at least one SPL. Each SPL in a benchmark must contain products that requires more than one round of model learning with respect to the basic active learning $L^*$ algorithm. Alternatively, tools supporting the synthesis of artificial benchmark models are also welcome.

## KEYWORDS

Model Learning, Behavioral Variability, Benchmarking, Featured Finite State Machines

**ACM Reference Format:**
Shaghayegh Tavassoli, Carlos Diego N. Damasceno, Mohammad Reza Mousavi, and Ramtin Khosravi. 2022. A Benchmark for Active Learning of Variability-Intensive Systems. In *26th International Systems and Software Product Line Conference - Volume A (SPLC '22), September 12-16, 2022, Graz, Austria.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Variability in a software system enables mass production and customization, particularly in the context of diverse and evolving requirements [12, 21]. To maximize these opportunities, one needs to analyze variability-intensive systems in a structured manner by considering variabilities and commonalities as important characteristics of such software systems. Family-based models are an enabler for such a structured analysis approach. Different types of family models have been used to analyze SPL functionality, including structural models [14, 16] and behavioral models [5, 9]. A type of structural model widely used in software product line engineering (SPLE) research is feature models [14, 16]. Structural models can be used to annotate traditional behavioral models to represent the behavior of a product line [5, 9]. Featured Transition System (FTS) [5] and Featured Finite State Machines (FFSM) [10, 11] are two examples of behavioral variability models for expressing the state-based behavior of families of software products. The FTS model extends traditional labelled transition systems with Boolean expressions for annotating the transitions. Likewise, the FFSM model is an extension of Mealy Finite State Machines (FSM) [13] that uses Boolean expressions on states and transitions as presence conditions [7, 11].

Active model learning is a process in which a model of software behavior is learned by presenting different inputs and observing the output [1, 8, 20]. Model learning can be used in the behavioral analysis of an SPL, for example in model-based testing and model checking [9]. When applied to variability-intensive systems, it is expected that the interactions with different variants or products will help to identify commonalities and variability. Adaptive model learning is a model learning approach in which previous models are reused to learn a new model, so that the speed of model learning can be increased [8]. In an SPL, the number of valid products may be exponential relative to the number of features [4, 9]. However, due to the commonalities of SPL products, it is possible to reuse existing models during the model learning of an SPL. Therefore, it may be possible to reduce the total cost of learning the SPL model by using these common features [9].

Active model learning is performed using two types of queries, namely membership queries (MQ) and equivalence queries (EQ). The number of EQs and MQs and the length of these queries can affect the efficiency of model learning [1, 8, 20]. To evaluate the model learning methods in the context of SPLs, it is necessary to have a benchmark set that can clearly show how efficiently and accurately different learning methods perform. In a suitable benchmark, it should be feasible to evaluate each learning cost metric (described in Section 3.4). Therefore, such benchmarks should contain products that require multiple rounds to be learned using usual non-adaptive methods to accurately show the effect of learning methods on models with different characteristics.


*SPLC '22, September 12-16, 2022, Graz, Austria*

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

## 2 PROBLEM DEFINITION

In this challenge, participants are asked to provide a benchmark set that can be used for evaluating model learning methods in variability-intensive systems. The benchmark set must contain at least one SPL. Each SPL in the benchmark set must contain at least one product which requires more than one iteration for model learning using the $L^*$ algorithm. The variability models and the behavioral models of these SPLs must be provided in the benchmark set. The characteristics that result in a multi-round model learning process should also be explained in the solution report. Alternatively, a tool for generating artificial SPL models can be provided as a solution. In this case, the tool must be able to generate random FFSMs or families of FSMs by taking the variability model and the input/output alphabets as input arguments.

## 3 BACKGROUND

In this section, some of the terms and definitions used in this paper are briefly explained. First, some well-known notations for structural and behavioral modeling of SPLs are described. Then, the model learning process and the metrics used to evaluate model learning methods are explained.

### 3.1 Feature Model

A feature model [14, 16] is a variability model that demonstrates the structure of an SPL as a hierarchical representation of SPL features. A feature model shows the commonality and variability of SPL products using a tree representation, implying the relationship between the features such that whenever a (non-root) feature exists in a configuration, its parent feature must exist in that configuration too. In this representation, mandatory features are specified using a solid circle. Optional features are represented using an empty circle. Alternative features are demonstrated as sibling nodes in the tree which are specified using an arc drawn between the edges of all options. If a product contains the parent feature, only one of the alternative features must be present in that product [9, 14, 16]. Feature dependencies can also be expressed using simple cross-tree constraints (i.e., *requires* and *excludes* relations) or using boolean logic expressions defined over the feature set [3].

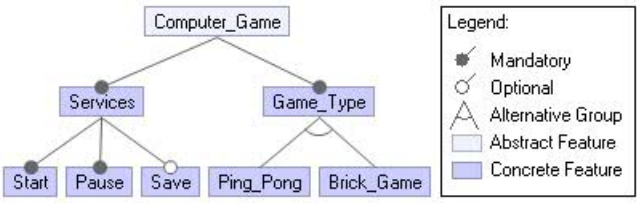


**Figure 1: The feature model of a simple computer game product line, inspired by [10]**

The feature model of a simple computer game is shown in Figure 1 (taken from [10] with minor modifications). In this figure, *Save* is an optional feature. *Services*, *Start*, *Pause* and *Game_Type* are mandatory features. In each valid product, exactly one feature from {*Ping_Pong*, *Brick_Game*} is present. Using a feature constraint, one can concisely specify one or more product configurations by expressing a boolean expression over the features. For example, the feature constraint ¬*Save* specifies the set of all product configurations which does not include the *Save* feature [14, 16].

### 3.2 Featured Behavioral Models

Various methods have been proposed for behavioral modeling of SPLs. Many of these notations are extensions of Finite State Machines (FSM) [13] or Labeled Transition Systems (LTS) [2] that can also incorporate features. Some notable examples are Featured Finite State Machines (FFSM) [10, 11] and Featured Transition Systems (FTS) [5, 6]. Given that most model learning methods aim at learning FSMs [20], in this section, we explain FFSM notation, which is an FSM-based featured behavioral model [10, 11]. The same concepts can be used to come up with benchmarks based on other family-based models such as FTSs.

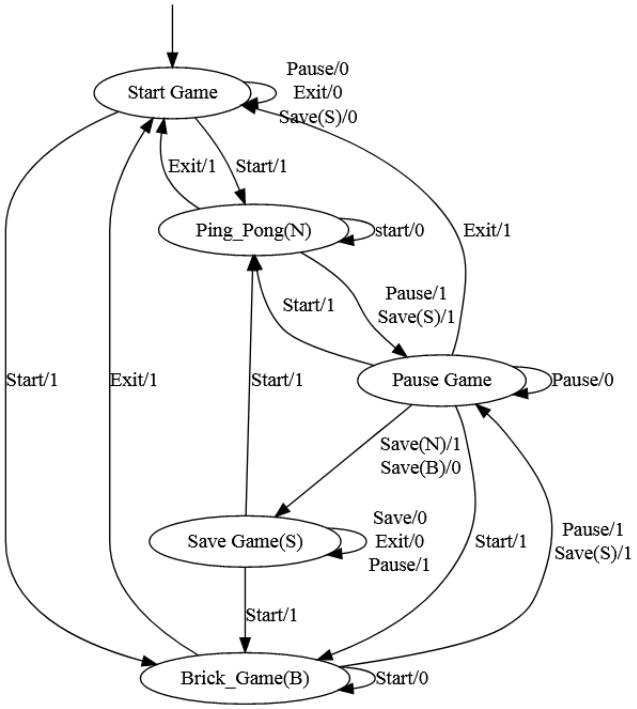


**Figure 2: The FFSM of a simple computer game product line, inspired by [10]**

A Finite State Machine (FSM) is a behavioral model and is defined as $M = \langle S, s_0, I, O, \delta, \lambda \rangle$. In this definition, $S$ is the set of FSM states and $s_0$ is its initial state. The set of input and output alphabets are represented by $I$ and $O$, respectively. The transition function is denoted by $\delta$ which determines what the next state is when the FSM is in state $s \in S$ and the input $a \in I$ is presented. The output function is denoted by $\lambda$ which maps each pair of a state and an input to an output. An FSM is deterministic if for each state and input, there exists at most one transition [8, 20].

A Featured Finite State Machine (FFSM) can be defined as an FSM whose states and transitions are labeled with feature constraints. An FFSM is defined as $(F, \Lambda, C, c_0, Y, O, \Gamma)$. In this definition, $F$ is

the set of features and $\Lambda$ is the set of valid configurations. The set of conditional states is represented by $C$. A conditional state is defined as $(s, b)$ where $s$ is a state and $b$ is a feature constraint. The initial conditional state is denoted by $c_0$ and is defined as $(s_0, true)$. The set of conditional inputs is denoted by $Y$ and the set of outputs is represented by $O$. Finally, $\Gamma$ specifies the set of conditional transitions. A conditional transition is defined as a tuple $(s_i, y, o, s_j)$ where $s_i$ and $s_j$ are conditional states, $y$ is a conditional input and $o$ is an output. An FFSM can be projected into the FSM of any given valid product using the product derivation operator [9–11].

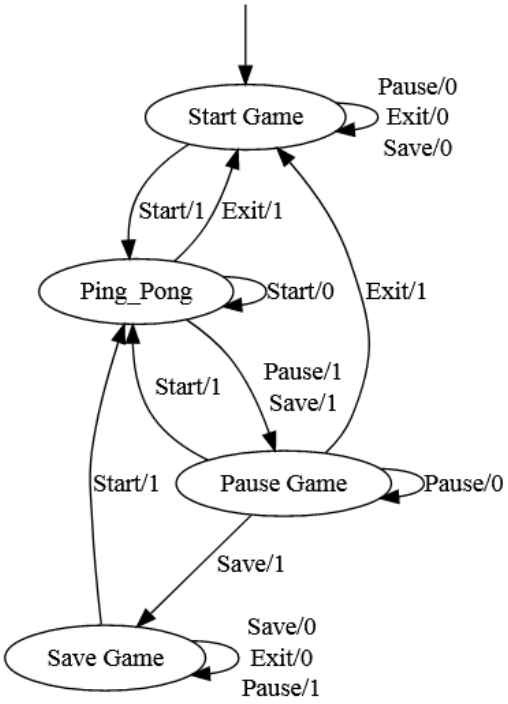


**Figure 3: The FSM of a product derived from the computer game product line, inspired by [10]**

The FFSM model of the computer game example described in 3.1 is shown in Figure 2. In this figure, the feature constraints of the states and inputs are written in capital letters in parentheses. The *Save* feature is shown with the letter $S$, the *Ping_Pong* feature with the letter $N$ and the *Brick_Game* feature with the letter $B$. Figure 3 shows the FSM of a product containing *Ping_Pong* and *Save* features derived from the FFSM in Figure 2 (Figures 2 and 3 are taken from [10] with minor modifications).

In this challenge, participants are asked to include, if possible, the FSMs of all valid SPL products in the benchmark set. If the number of valid products is large, a featured behavioral model should be provided (instead of the FSMs of all products). In this case, it is preferred to provide a tool that can be used to derive the FSM of any given valid configuration using the featured behavioral model.

## 3.3 Model Learning

Model learning [20] is a method for constructing a state machine that represents the behavior of a software system. During the active model learning process, different input sequences are presented to the system under learning (SUL) and its output is observed. Based on the observed results, a hypothesis about the SUL behavior (the FSM language) is constructed. In the $L^*$ algorithm [1] proposed by Dana Angluin, two types of queries are used to learn the behavior of the SUL (a deterministic FSM). Using a membership query (MQ), it is asked what the SUL output is for a given input sequence. This type of queries is used in the hypothesis construction phase of learning. Using an equivalence query (EQ) it is asked whether hypothesis $H$ about the SUL language is correct. If the hypothesis is correct, the learning process will be terminated. If the hypothesis is incorrect, a counterexample is provided in which the result of $H$ differs from the SUL language. Equivalence queries are used in the hypothesis validation phase of model learning [1, 8, 20].

During the model learning process, the query results are stored in an observation table. An observation table $OT$ can be defined as a triple $(S, E, T)$, where $S$ is a set of prefixes which is prefix-closed. A prefix-closed set of strings is a set that contains all the prefixes of all of its members. In the definition of the observation table, $E$ is a set of suffixes. Assuming $s \in S$ and $e \in E$, $T[s, e]$ is the last output of the SUL when the input sequence is $s.e$ (i.e., $s$ concatenated with $e$). In the $L^*$ algorithm, when a counterexample $c$ is returned, $c$ and its prefixes are added to $S$ in the observation table and the new values of $T$ are calculated using MQs. This refined observation table will be used in the next iteration of the algorithm [18]. While the $L^*$ algorithm was originally designed for inferring deterministic finite automata, we focus on the variant tailored to Mealy machines [8, 17].

## 3.4 Metrics

The time complexity of model learning algorithms depends on the number of EQs and MQs used in the learning process. In each iteration of $L^*$, one equivalence query (conjecture) is used. Therefore, the number of iterations (rounds) can be considered as an important learning cost metric [1, 18, 20]. Another parameter that affects the time complexity of learning algorithms, is the length of EQs and MQs. The total number of symbols used in MQs and in the implementation of EQs can be used as another cost metric [20]. Also, the number of reset operations can affect the cost of learning [8]. To evaluate the efficiency of model learning methods, it is necessary to be able to compare the values of these metrics in different learning methods. A suitable benchmark set and its associated metrics can be very useful when evaluating the model learning methods.

# 4 THE CHALLENGE

In this challenge, participants are asked to provide a benchmark set that can be used to evaluate model learning methods in the software product line context.

Alternatively, a tool support for generating artificial SPLs, i.e., *random FFSMs* or *family of FSMs randomly generated* for a given variability model; is also welcome.

### 4.1 Content of the Challenge Solution

The benchmark set must contain the following items for at least one SPL:

(1) **Variability model:** As a variability model, the SPL feature model must be provided. The feature model must be in the XML format of the FeatureIDE [19] library.
(2) **Behavioral model:** To demonstrate the behavior of the SPL, one of the following methods must be used:
   - If possible, the FSM of each valid configuration of the SPL must be provided in the format of a dot file.
   - If the number of valid products of an SPL is large, the following items must be provided:
     (a) A featured behavioral model representing the behavior of the SPL
     (b) A derivation tool that can derive the FSM of any given valid configuration using the featured behavioral model of the SPL (the derived FSMs must be in dot format)

**Random FFSM/FSM model generator.** Alternatively, participants can submit methods to generate artificial SPLs in the format of random families of FSMs (or FFSMs), with their respective implementations. A sample of artificial models produced with their method must be provided according to the requirements previously discussed.

The methods submitted must allow users to define the following parameters:

(1) The variability source (i.e., feature model)
(2) The randomness source (i.e., seed)
(3) The number of states of the generated FSMs/FFSM
(4) The set of input/output symbols (i.e., I/O alphabets)

### 4.2 Characteristics of the Benchmark Set

This benchmark set should be applicable to evaluate the efficiency, conciseness, and accuracy of model learning methods in the SPL context. Using this benchmark set, it should be possible to compare model learning cost metrics, such as EQs, MQs, the number of rounds, conciseness, and accuracy. The metrics should show some reduction in the cost of learning the family model (e.g., in the number of MQs, EQs, or the total number of symbols) with respect to learning the individual products. Two other important metrics to be evaluated using the benchmark-set are the conciseness [7] of the family model and its accuracy [9]. The learned family model is supposed to merge the commonalities into common states and transitions and hence, ought to be much smaller than the sum of the size of the individual products. Moreover, the learned family model should be accurate in the sense that all product behaviors derived from the learned family model should be equivalent to the products of the original product-line under learning. To enable measuring these two metrics, we need to have access to the original (possibly hand-crafted) models of the product lines and a tool for deriving product models from the family model.

Each SPL in this benchmark set must contain at least one configuration (SPL product) that requires more than one round for its model to be learned using the $L^*$ algorithm. To measure the learning cost metrics, the learning experiments have to be performed using LearnLib [15] library version 0.16.0[1].

The non-adaptive model learning can be performed using ExtensibleLStarMealyBuilder class from the LearnLib library [2] using the following parameters:

- The set of initial prefixes is $S = \{\epsilon\}$.
- The set of initial suffixes is $E = I$, where $I$ is the input alphabet.
- The values of other learning parameters are optional. These parameters include:
  – The equivalence oracle type
  – The counterexample handling method
  – The observation table closing strategy
  – Whether caching is used or not

  The assumed values for these parameters should be explained in the solution report.

Participants are also asked to explain the reasons why some product models require more than one round to be learned. The characteristics of such products can be explained in the solution report.

The detailed information for providing the benchmark set is available on GitHub[3]. In this repository, there is a Java class called *AutomataLearning* with a code example for experimenting with the LearnLib framework [4] for active model learning [15]. This class takes an FSM file in dot format as an input. The model of this FSM is learned using the $L^*$ algorithm and the values of the learning cost metrics are printed.

## 5 SUMMARY

In this challenge, participants are asked to provide a benchmark set that can be used for evaluating model learning methods in variability-intensive systems. This benchmark set should be applicable for comparing the efficiency of different model learning methods in the SPL context.

The provided benchmark set must contain at least one SPL, in which at least one of its products requires more than one round for its model to be learned using the $L^*$ method. The benchmark set should contain the variability models and the behavioral models of the SPLs. Alternatively, a tool for generating families of artificial FSMs/FFSMs can be provided, in combination with a sample of random behavioral variability models. The participants are also asked to explain the reasons why some product models require more than one round for model learning. The submitted solutions will be evaluated using the requirements and the experimental setup explained in section 4.2. The set of accepted solutions can be used in the future as benchmarks for evaluating model learning methods in variability-intensive systems.

[1] https://github.com/LearnLib/learnlib/tree/learnlib-0.16.0
[2] http://learnlib.github.io/learnlib/maven-site/0.14.0/apidocs/de/learnlib/algorithms/lstar/mealy/ExtensibleLStarMealyBuilder.html
[3] https://github.com/damascenodiego/splc22challenge_dataset_learning
[4] https://learnlib.de/

## ACKNOWLEDGMENTS

Mohammad Reza Mousavi has been partially supported by the UKRI Trustworthy Autonomous Systems Node in Verifiability, Grant Award Reference EP/V026801/1.

## REFERENCES

[1] Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.* 75, 2 (1987), 87–106. https://doi.org/10.1016/0890-5401(87)90052-6

[2] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking.* MIT Press.

[3] Don S. Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Software Product Lines, 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005, Proceedings (LNCS, Vol. 3714)*, J. Henk Obbink and Klaus Pohl (Eds.). Springer, 7–20. https://doi.org/10.1007/11554844_3

[4] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Trans. Software Eng.* 39, 12 (2013), 1611–1640. https://doi.org/10.1109/TSE.2013.34

[5] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-François Raskin. 2013. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Trans. Software Eng.* 39, 8 (2013), 1069–1089. https://doi.org/10.1109/TSE.2012.86

[6] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. 2010. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010.* ACM, 335–344. https://doi.org/10.1145/1806799.1806850

[7] Carlos Diego Nascimento Damasceno, Mohammad Reza Mousavi, and Adenilso da Silva Simão. 2019. Learning from difference: an automated approach for learning family models from software product lines. In *Proceedings of the 23rd International Systems and Software Product Line Conference, SPLC 2019, Volume A, Paris, France, September 9-13, 2019.* ACM, 10:1–10:12. https://doi.org/10.1145/3336294.3336307

[8] Carlos Diego Nascimento Damasceno, Mohammad Reza Mousavi, and Adenilso da Silva Simão. 2019. Learning to Reuse: Adaptive Model Learning for Evolving Systems. In *Integrated Formal Methods - 15th International Conference, IFM 2019, Bergen, Norway, December 2-6, 2019, Proceedings (LNCS, Vol. 11918)*, Wolfgang Ahrendt and Silvia Lizeth Tapia Tarifa (Eds.). Springer, 138–156. https://doi.org/10.1007/978-3-030-34968-4_8

[9] Carlos Diego Nascimento Damasceno, Mohammad Reza Mousavi, and Adenilso da Silva Simão. 2021. Learning by sampling: learning behavioral family models from software product lines. *Empir. Softw. Eng.* 26, 1 (2021), 4. https://doi.org/10.1007/s10664-020-09912-w

[10] Vanderson H. Fragal, Adenilso Simão, and Mohammad Reza Mousavi. 2016. Validated Test Models for Software Product Lines: Featured Finite State Machines. In *Formal Aspects of Component Software - 13th International Conference, FACS 2016, Besançon, France, October 19-21, 2016, Revised Selected Papers (LNCS, Vol. 10231)*, Olga Kouchnarenko and Ramtin Khosravi (Eds.). 210–227. https://doi.org/10.1007/978-3-319-57666-4_13

[11] Vanderson Hafemann Fragal, Adenilso Simão, Mohammad Reza Mousavi, and Uraz Cengiz Türker. 2019. Extending HSI Test Generation Method for Software Product Lines. *Comput. J.* 62, 1 (2019), 109–129. https://doi.org/10.1093/comjnl/bxy046

[12] Matthias Galster, Danny Weyns, Dan Tofan, Bartosz Michalik, and Paris Avgeriou. 2014. Variability in Software Systems - A Systematic Literature Review. *IEEE Trans. Software Eng.* 40, 3 (2014), 282–306. https://doi.org/10.1109/TSE.2013.56

[13] Arthur Gill et al. 1962. Introduction to the theory of finite-state machines. (1962).

[14] 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 Univ Pittsburgh Pa Software Engineering Inst.

[15] Harald Raffelt and Bernhard Steffen. 2006. LearnLib: A Library for Automata Learning and Experimentation. In *Proc. of the 9th International Conference on Fundamental Approaches to Software Engineering (FASE 2006) (LNCS, Vol. 3922).* Springer, 377–380. https://doi.org/10.1007/11693017_28

[16] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. 2006. Feature Diagrams: A Survey and a Formal Semantics. In *14th IEEE International Conference on Requirements Engineering (RE 2006), 11-15 September 2006, Minneapolis/St.Paul, Minnesota, USA.* IEEE Computer Society, 136–145. https://doi.org/10.1109/RE.2006.23

[17] Muzammil Shahbaz and Roland Groz. 2009. Inferring Mealy Machines. In *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5850)*, Ana Cavalcanti and Dennis Dams (Eds.). Springer, 207–222. https://doi.org/10.1007/978-3-642-05089-3_14

[18] Bernhard Steffen, Falk Howar, and Maik Merten. 2011. Introduction to Active Automata Learning from a Practical Perspective. In *Formal Methods for Eternal Networked Software Systems - 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures (LNCS, Vol. 6659)*, Marco Bernardo and Valérie Issarny (Eds.). Springer, 256–296. https://doi.org/10.1007/978-3-642-21455-4_8

[19] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Sci. Comput. Program.* 79 (2014), 70–85. https://doi.org/10.1016/j.scico.2012.06.002

[20] Frits W. Vaandrager. 2017. Model learning. *Commun. ACM* 60, 2 (2017), 86–95. https://doi.org/10.1145/2967606

[21] Jilles van Gurp, Jan Bosch, and Mikael Svahnberg. 2001. On the Notion of Variability in Software Product Lines. In *2001 Working IEEE / IFIP Conference on Software Architecture (WICSA 2001), 28-31 August 2001, Amsterdam, The Netherlands.* IEEE Computer Society, 45–54. https://doi.org/10.1109/WICSA.2001.948406