Static Analysis and Family-based Model Checking of Featured Transition Systems with VMC

Maurice H. ter Beek
Franco Mazzanti
ISTI–CNR, Pisa, Italy

Ferruccio Damiani
Luca Paolini
Giordano Scarso
Michele Valfrè
University of Turin, Italy

Michael Lienhardt
ONERA, Palaiseau, France

ABSTRACT

A Featured Transition System (FTS) is a formalism for modeling variability in configurable system behavior. The behavior of all variants (products) is modeled in a single compact FTS by associating the possibility to perform an action and transition from one state to another with feature expressions that condition the execution of an action in specific variants. We present a front-end for the research tool VMC. The resulting toolchain allows a modeler to analyze an FTS for ambiguities (dead or false optional transitions and hidden deadlock states), transform an ambiguous FTS into an unambiguous one, and perform an efficient kind of family-based verification of an FTS without hidden deadlock states. We use benchmarks from the literature to demonstrate the novelties offered by the toolchain.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; **Formal methods**; **Model checking**; **Automated static analysis**.

KEYWORDS

SPL, variability, FTS, MTS, static analysis, formal verification, VMC

ACM Reference Format:
Maurice H. ter Beek, Franco Mazzanti, Ferruccio Damiani, Luca Paolini, Giordano Scarso, Michele Valfrè, and Michael Lienhardt. 2021. Static Analysis and Family-based Model Checking of Featured Transition Systems with VMC. In *25th ACM International Systems and Software Product Line Conference - Volume B (SPLC '21), September 6–11, 2021, Leicester, United Kingdom.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3461002.3473071

1 INTRODUCTION AND BACKGROUND

The automated analysis of variability models, such as the detection of anomalies like dead or false optional features in feature diagrams, has a 30-year history [13, 29]. Variability in behavioral models has a shorter history [23, 24, 26] Moreover, such behavioral models received considerable attention only during the last decade, following the seminal paper by Classen et al. [18] that introduced FTSs and an efficient means to model check them.

SPLC '21, September 6–11, 2021, Leicester, United Kingdom

ACM ISBN 978-1-4503-8470-4/21/09…$15.00
https://doi.org/10.1145/3461002.3473071

An FTS is a formal model with variability encoding to capture the behavior of all variants of a configurable system in a single transition system [17]; its transitions, labeled with actions, are associated with *feature expressions* that condition their presence in classical labeled Transition Systems (LTSs) that model individual variant behavior. Proving correctness of such behavioral models through model checking is challenging. Ideally, the compactness of the FTS is exploited to reason on the whole system at once. Such an *all-in-one* technique, by which the behavior of all variants is examined only once simultaneously, is called family-based analysis, in contrast to an enumerative product-based analysis, by which the behavior of each individual variant is examined *one-by-one* [28]. During the past decade, FTSs proved amenable to family-based model-checking [6, 12, 15–17, 19–21]

In [3, 5], we tackled the automated analysis of FTSs. We defined three ambiguities for an FTS: (i) a *dead transition* (i.e. a transition that is unreachable, and thus cannot be executed, in any variant); (ii) a *false optional transition* (i.e. a transition that can be executed in all variants in which its source state is reachable); and (iii) a *hidden deadlock state* (i.e. a state from which a transition can be executed only in some variants, but not all). In analogy with the above mentioned anomaly detection for variability models, we developed an algorithm to detect ambiguities in FTSs, and a means to resolve them, with a proof of its correctness. Anomalies are often due to an incorrect use of cross-tree constraints and solving them typically means removing a transition or correcting a feature expression. An ambiguous FTS is often undesired, since it gives an unclear view of the behavior of the configurable system. Moreover, an unambiguous FTS paves the way for an efficient kind of family-based model checking. We implemented this algorithm exploiting the SAT solving features of the Z3 SMT solver. The Python code of our implementation (analyzer.py), which accepts FTSs in the format .dot as input, is publicly available [4].

In this paper, we present FTS4VMC, a front-end for the research tool VMC [7, 9, 11], developed to make VMC amenable to FTSs. VMC (http://fmt.isti.cnr.it/vmc) is a tool for the analysis of behavioral models with variability during a system's early design phase. It accepts as input a Modal Transition System (MTS) with a set of logical *variability constraints* (MTSv), akin to an FTS' feature expressions. An MTS [25] is an LTS that distinguishes *admissible* ('may'), *necessary* ('must'), and *optional* (may but not must) transitions which are such that by definition all necessary and optional transitions are also admissible. MTSs were introduced to capture the refinement of a partial description into a more detailed one, reflecting increased knowledge on the admissible (but not necessary)

behavior. MTSυs were introduced in [7] to compactly model SPL behavior, whose individual variant behavior, in the form of an LTS, can be obtained by means of a special-purpose refinement relation, or by an equivalent operational derivation procedure. In [2], it was shown that such MTSs are equally expressive as FTSs.

The research tool VMC offers explicit-state on-the-fly model checking of MTS properties expressed in the dedicated variability-aware action-based and state-based branching-time temporal logic v-ACTL [1, 7] derived from ACTL, which is an action-based version of the well-known logic CTL. VMC offers both product-based analysis, upon explicit generation of behavioral variant models (LTSs), and a kind of family-based analysis (on MTSs). In [7], a specific kind of family-based model checking was introduced for MTSυs, which we explain next. First, an important safety property is *deadlock freedom*, i.e. a system should not deadlock by reaching a (deadlock) state in which no further action is possible, thus guaranteeing progress or liveness. In case of configurable systems (like FTSs or MTSs) this notion can be extended to guaranteeing liveness for each variant (LTS). Now, an MTSυ is defined to be *live* if all its states are live, where a live state of an MTSυ is such that it does not occur as a deadlock state in any of its variants, effectively resulting in an MTSυ in which every path is infinite. It was proved that for properties expressed in v-ACTLive$^{\Box}$, which is a rich fragment of v-ACTL interpreted on live MTSs, validity on the (live) MTS guarantees validity of the property for all its variants (cf. [7, Theorem 4]), thus allowing a kind of family-based model checking of MTSυs. In [5], it was shown that this result continues to hold for MTSυs whose every state is either live or a deadlock.

In [3, 5], we noted that any FTS $\mathcal{F}$ can be transformed into an MTS $\mathcal{F}_{\text{MTS}}$ by considering its must transitions as necessary transitions, its featured transitions as optional transitions, and all its transitions as admissible, and by removing all feature expressions. If $\mathcal{F}$ is live, then $\mathcal{F}_{\text{MTS}}$ is live, with respect to the FTS's set of variants $\text{lts}(\mathcal{F})$, because it has no hidden deadlocks. Moreover, all transitions of $\mathcal{F}$ whose corresponding (LTS) transitions are mandatorily present in all variants, correspond to necessary transitions in $\mathcal{F}_{\text{MTS}}$. This demonstrates that the above result from [7] can be carried over to live FTSs, thus allowing a kind of family-based model checking also on such FTSs for the v-ACTL fragment v-ACTLive$^{\Box}$. Hence, any formula ϕ of v-ACTLive$^{\Box}$ is preserved by live FTSs: given a live FTS $\mathcal{F}$, whenever ϕ holds for $\mathcal{F}_{\text{MTS}}$, denoted by $\mathcal{F}_{\text{MTS}} \models \phi$, then ϕ holds for all variants $\mathcal{L} \in \text{lts}(\mathcal{F})$ of $\mathcal{F}$, i.e. $\mathcal{L} \models \phi$.

More precisely, if (i) the FTS is live, which is the case if it has no hidden deadlocks (so, unambiguous FTSs are live), and (ii) the property ϕ to be verified is specified in v-ACTLive$^{\Box}$, then ϕ can be verified directly on the FTS (ignoring its feature expressions) and if (iii) ϕ holds, then validity is preserved in all LTSs modeling variant behavior, i.e. ϕ holds for all variants. If any of these three conditions does not hold, the property needs to be verified with classical (family-based) approaches, like those mentioned in Section 2.

The newly developed front-end FTS4VMC for VMC allows a modeler (i) to check an FTS for ambiguities (dead or false optional transitions and hidden deadlock states), by calling analyzer.py; (ii) to remove ambiguities and thus turn an ambiguous FTS into an unambiguous one, by calling disambiguator.py; (iii) to transform an FTS into an MTS, by calling translator.py; and (iv) to perform an efficient kind of family-based model checking of an MTS obtained as described in iii from an FTS without hidden deadlock states, by calling VMC via vmc_controller.py. The FTS4VMC implementation, including the Python code of disambiguator.py and translator.py, is publicly available from https://github.com/fts4vmc/FTS4VMC.

We use benchmarks from the literature to demonstrate the novelties offered by the resulting toolchain (cf. also the tutorial [10]).

2 RELATED WORK

An encompassing overview of SPL analysis strategies, including static analysis and model checking, can be found in [28] and a recent empirical study on applying variability-aware static analysis techniques to real-world configurable systems is presented in [27]. Static analysis of FTSs mimics feature-model analysis by defining behavioral counterparts of dead and false optional features [13, 29].

Family-based model checking of behavioral variability models provides a means to simultaneously verify multiple behavioral variant models in a single run. Properties can be verified with dedicated SPL model-checking tools like SNIP [15, 17], ProVeLines [19], VMC [7, 9, 11], ProFeat [14] (for probabilistic model checking), or QFLan [8, 30] (for statistical model checking), or—by suitable abstractions or encodings—with well-known classical model checkers like PRISM [22] (for probabilistic model checking), mCRL2 [6, 12], SPIN [21], or NuSMV [20]. The survey [28] also discusses software model checking, operating directly on source code in Java or C.

3 TOOLCHAIN AT WORK

The methodology outlined in the Introduction has been fully automated. With the newly developed front-end FTS4VMC, the toolchain constituted by (i) FTS4VMC, (ii) the Python code analyzer.py from [4] (implementing the static analysis algorithm from [5], where it was shown to be more efficient than the one presented in [3]) and (iii) VMC, can assist an end user (modeler) in the following steps of the engineering and verification methodology envisioned in Fig. 1 (in which all green blocks have been automated by the toolchain):

- specify or upload an FTS in FTS4VMC or an MTSυ in VMC;
- use FTS4VMC to check whether the FTS is ambiguous (by calling analyzer.py);
- use FTS4VMC to transform any ambiguous FTS into an unambiguous one (or to remove only some particular kinds of ambiguities detected, by calling disambiguator.py);
- decide whether the FTS is live with FTS4VMC or whether the MTSυ is live with VMC;
- specify a v-ACTL formula in FTS4VMC or in VMC;
- decide whether the v-ACTL formula is a v-ACTLive$^{\Box}$ formula with VMC;
- verify a v-ACTLive$^{\Box}$ formula on the FTS (transformed into an MTS by FTS4VMC, by calling translator.py) or directly on the MTSυ (transformed into an MTS by VMC) with VMC;
- report validity for all variants of the FTS/MTSυ in case a v-ACTLive$^{\Box}$ formula was verified to hold on a live FTS/MTSυ (a kind of family-based model checking) with VMC;
- verify a v-ACTLive$^{\Box}$/v-ACTL formula on all variants (generated by VMC) of the MTSυ (product-based model checking) with VMC.

The novelties are clearly indicated in the figure: the blue steps and the green steps if applied to FTSs are made possible by FTS4VMC.

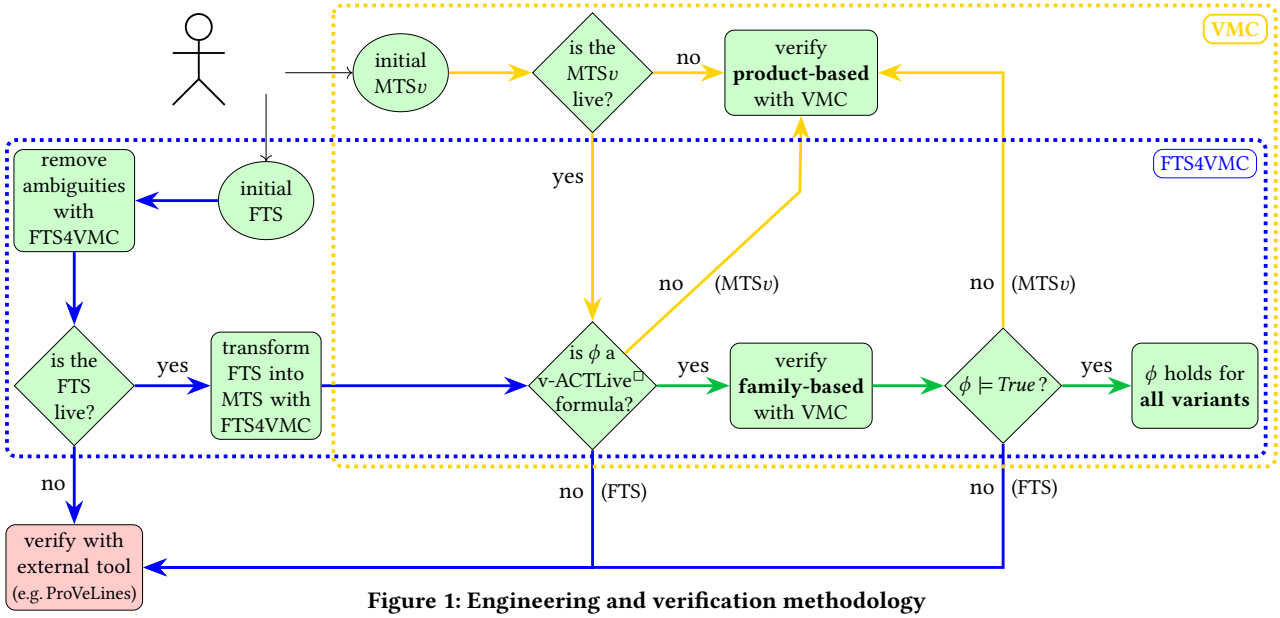

Figure 1: Engineering and verification methodology

The front-end FTS4VMC accepts FTSs in the .dot format, a well-known graphical notation supported by the graphviz open source graph visualization software (cf. https://www.graphviz.org).

Once the FTS has been uploaded and verified to be in the correct format, the user (modeler) is offered a variety of (analysis) tasks:

Full ambiguities analysis analyzes the FTS for all three types of ambiguities: once done, it outputs an updated version of the FTS with dead transitions highlighted in blue, false optional transitions highlighted in green, and hidden deadlock states highlighted in red; moreover, it enables the ambiguity removal tasks mentioned below;

Liveness analysis analyzes the FTS for liveness, i.e. whether it has hidden deadlock states but ignoring the detection of dead and false optional transitions;

Stop processing interrupts the current (full ambiguities or liveness) analysis;

Remove all ambiguities performs the next two analyses at once;

Remove false optional transitions replaces the feature expression of each false optional transition of the FTS with *True*, thus converting these transitions into must transitions; no transitions are added or deleted;

Remove dead transitions + hidden deadlock states deletes unreachable transitions if present, then resolves hidden deadlocks;

View MTS shows the MTS obtained from the FTS by turning each must transition (i.e. with feature expression *True*) into a necessary one and making every other transition admissible; updates source and graph;

Verify property verifies properties expressed in v-ACTL with VMC, once the analysis was completed and the FTS results live, by inserting them in the provided text area;

Show explanation shows the counterexample provided by VMC after having checked the property.

During execution of these tasks, the user is offered different views:

Console displays progress and results of the performed tasks;

Source displays the .dot source file of the current FTS;

Graph displays the rendered graph in SVG format, highlighting the feature model and ambiguities;

Summary displays the console output after successful analysis in a more user-friendly way;

Counterexample graph displays the counterexample obtained upon interaction with VMC rendered as a graph.

Moreover, at any time, the user can download the displayed result (e.g. the FTS in .dot or SVG formal and with or without highlighted ambiguities, the transformed MTS, etc.).

4 FUTURE WORK

In [3], we considered branching-time CTL-like properties. In [5], we also considered linear-time LTL-like properties and showed that a live FTS enjoys the property that all valid linear-time LTL formulas are preserved by all its variants. This can be seen as follows. A path in an LTS is said to be *maximal* if it cannot be extended further, i.e. it is infinite or it ends in a deadlock state. Model checking LTL formulas on an LTS reduces to analyzing its maximal paths: an LTL formula is valid if it holds for all maximal paths. These notions trivially carry over to FTSs by ignoring their feature expressions. Clearly, if an FTS is live, i.e. it has no hidden deadlocks, then the set of maximal paths of any variant (LTS) is a subset of the set of maximal paths of the FTS. Hence, the following result holds. Any formula ϕ of LTL is preserved by live FTSs: given a live FTS $\mathcal{F}$, whenever ϕ holds for $\mathcal{F}_{\text{LTS}}$, denoted by $\mathcal{F}_{\text{LTS}} \models \phi$, then ϕ holds for all variants $\mathcal{L} \in \text{lts}(\mathcal{F})$ of $\mathcal{F}$, i.e. $\mathcal{L} \models \phi$. We thus envision the verification methodology depicted in Fig. 2.

5 CONCLUSION

We presented FTS4VMC, developed specifically as a front-end for the research tool VMC, with a user-friendly GUI. It has code to

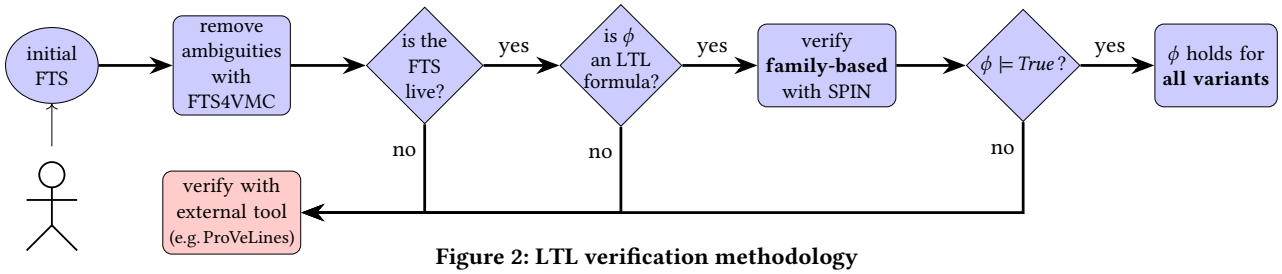

Figure 2: LTL verification methodology

upload and download files, handle users' session data, render graph visualization and HTML output, and communicate with VMC. It also contains Python code: analyzer.py from [4], implementing the ambiguities analysis; disambiguator.py, implementing the ambiguities removal; graph.py, implementing the FTS/MTS graph rendering; translator.py, implementing the transformation of an FTS into an MTS; vmc_controller.py, handling the property verification with VMC; and process_manager.py, handling multiprocessing required for real-time output during the analysis process. The resulting toolchain allows a modeler to analyse an FTS for ambiguities, remove them, and perform an efficient kind of family-based model checking of specific branching-time properties on the resulting FTS.

The results, mentioned in this paper, that form the basis of the verification methodologies depicted in Figs. 1 and 2 indicate specific cases in which verification of live FTSs reduces to verification (with a linear complexity) of corresponding MTSs and LTSs that are obtained straightforwardly by ignoring the feature expressions (and distinguishing necessary and optional transitions in case of MTSs). However, if either (i) the property to be verified is not a v-ACTLive$^{\square}$ or LTL formula, or (ii) the result of the verification is false, then the formula needs to be verified with a classical family-based model-checking tool or by means of product-based model checking.

REFERENCES

[1] M.H. ter Beek, F. Damiani, S. Gnesi, F. Mazzanti, and L. Paolini. 2015. From Featured Transition Systems to Modal Transition Systems with Variability Constraints. In *SEFM'15 (LNCS, Vol. 9276)*. Springer, 344–359. https://doi.org/10.1007/978-3-319-22969-0_24

[2] M.H. ter Beek, F. Damiani, S. Gnesi, F. Mazzanti, and L. Paolini. 2019. On the Expressiveness of Modal Transition Systems with Variability Constraints. *Sci. Comput. Program.* 169 (2019), 1–17. https://doi.org/10.1016/j.scico.2018.09.006

[3] M.H. ter Beek, F. Damiani, M. Lienhardt, F. Mazzanti, and L. Paolini. 2019. Static Analysis of Featured Transition Systems. In *SPLC'19*. ACM, 39–51. https://doi.org/10.1145/3336294.3336295

[4] M.H. ter Beek, F. Damiani, M. Lienhardt, F. Mazzanti, and L. Paolini. 2019. Supplementary material for: "Static Analysis of Featured Transition Systems". https://doi.org/10.5281/zenodo.2616646

[5] M.H. ter Beek, F. Damiani, M. Lienhardt, F. Mazzanti, and L. Paolini. 2021. Efficient Static Analysis and Verification of Featured Transition Systems. *Empir. Softw. Eng.* (2021). https://doi.org/10.1007/s10664-020-09930-8

[6] M.H. ter Beek, E.P. de Vink, and T.A.C. Willemse. 2017. Family-Based Model Checking with mCRL2. In *FASE'17 (LNCS, Vol. 10202)*. Springer, 387–405. https://doi.org/10.1007/978-3-662-54494-5_23

[7] M.H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. 2016. Modelling and analysing variability in product families: Model checking of modal transition systems with variability constraints. *J. Log. Algebr. Meth. Program.* 85, 2 (2016), 287–315. https://doi.org/10.1016/j.jlamp.2015.11.006

[8] M.H. ter Beek, A. Legay, A. Lluch Lafuente, and A. Vandin. 2020. A framework for quantitative modeling and analysis of highly (re)configurable systems. *IEEE Trans. Softw. Eng.* 46, 3 (2020), 321–345. https://doi.org/10.1109/TSE.2018.2853726

[9] M.H. ter Beek and F. Mazzanti. 2014. VMC: Recent Advances and Challenges Ahead. In *SPLC'14*, Vol. 2. ACM, 70–77. https://doi.org/10.1145/2647908.2655969

[10] M.H. ter Beek, F. Mazzanti, F. Damiani, L. Paolini, G. Scarso, and M. Lienhardt. 2021. Static Analysis and Family-based Model Checking with VMC. In *SPLC'21*. ACM. https://doi.org/10.1145/3461001.3472732

[11] M.H. ter Beek, F. Mazzanti, and A. Sulova. 2012. VMC: A Tool for Product Variability Analysis. In *FM'12 (LNCS, Vol. 7436)*. Springer, 450–454. https://doi.org/10.1007/978-3-642-32759-9_36

[12] M.H. ter Beek, S. van Loo, E.P. de Vink, and T.A.C. Willemse. 2020. Family-Based SPL Model Checking Using Parity Games with Variability. In *FASE'20 (LNCS, Vol. 12076)*. Springer, 245–265. https://doi.org/10.1007/978-3-030-45234-6_12

[13] D. Benavides, S. Segura, and A. Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: a Literature Review. *Inf. Syst.* 35, 6 (2010), 615–636. https://doi.org/10.1016/j.is.2010.01.001

[14] P. Chrszon, C. Dubslaff, S. Klüppelholz, and C. Baier. 2018. ProFeat: feature-oriented engineering for family-based probabilistic model checking. *Form. Asp. Comp.* 30, 1 (2018), 45–75. https://doi.org/10.1007/s00165-017-0432-4

[15] A. Classen, M. Cordy, P. Heymans, A. Legay, and P.Y. Schobbens. 2012. Model checking software product lines with SNIP. *Int. J. Softw. Tools Technol. Transf.* 14, 5 (2012), 589–612. https://doi.org/10.1007/s10009-012-0234-1

[16] A. Classen, M. Cordy, P. Heymans, A. Legay, and P.Y. Schobbens. 2014. Formal semantics, modular specification, and symbolic verification of product-line behaviour. *Sci. Comput. Program.* 80, B (2014), 416–439. https://doi.org/10.1016/j.scico.2013.09.019

[17] A. Classen, M. Cordy, P.Y. Schobbens, P. Heymans, A. Legay, and J.F. Raskin. 2013. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Trans. Softw. Eng.* 39, 8 (2013), 1069–1089. https://doi.org/10.1109/TSE.2012.86

[18] A. Classen, P. Heymans, P.Y. Schobbens, A. Legay, and J.F. Raskin. 2010. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In *ICSE'10*. ACM, 335–344. https://doi.org/10.1145/1806799.1806850

[19] M. Cordy, A. Classen, P. Heymans, P.Y. Schobbens, and A. Legay. 2013. ProVeLines: A Product Line of Verifiers for Software Product Lines. In *SPLC'13*, Vol. 2. ACM, 141–146. https://doi.org/10.1145/2499777.2499781

[20] A.S. Dimovski. 2020. CTL* family-based model checking using variability abstractions and modal transition systems. *Int. J. Softw. Tools Technol. Transf.* 22, 1 (2020), 35–55. https://doi.org/10.1007/s10009-019-00528-0

[21] A.S. Dimovski, A.S. Al-Sibahi, C. Brabrand, and A. Wąsowski. 2017. Efficient family-based model checking via variability abstractions. *Int. J. Softw. Tools Technol. Transf.* 5, 19 (2017), 585–603. https://doi.org/10.1007/s10009-016-0425-2

[22] C. Dubslaff, C. Baier, and S. Klüppelholz. 2015. Probabilistic Model Checking for Feature-Oriented Systems. In *Transactions on AOSD XII (LNCS, Vol. 8989)*. Springer, 180–220. https://doi.org/10.1007/978-3-662-46734-3_5

[23] A. Fantechi and S. Gnesi. 2008. Formal Modeling for Product Families Engineering. In *SPLC'08*. IEEE, 193–202. https://doi.org/10.1109/SPLC.2008.45

[24] A. Gruler, M. Leucker, and K. D. Scheidemann. 2008. Modeling and Model Checking Software Product Lines. In *FMOODS'08 (LNCS, Vol. 5051)*. Springer, 113–131. https://doi.org/10.1007/978-3-540-68863-1_8

[25] J. Křetínský. 2017. 30 Years of Modal Transition Systems: Survey of Extensions and Analysis. In *Models, Algorithms, Logics and Tools*. LNCS, Vol. 10460. Springer, 36–74. https://doi.org/10.1007/978-3-319-63121-9_3

[26] K. Lauenroth, K. Pohl, and S. Töhning. 2009. Model Checking of Domain Artifacts in Product Line Engineering. In *ASE'09*. IEEE, 269–280. https://doi.org/10.1109/ASE.2009.16

[27] A. von Rhein, J. Liebig, A. J.ker, C. Kästner, and S. Apel. 2018. Variability-Aware Static Analysis at Scale: An Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 27, 4 (2018), 18:1–18:33. https://doi.org/10.1145/3280986

[28] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1 (2014), 6:1–6:45. https://doi.org/10.1145/2580950

[29] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. 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

[30] A. Vandin, M.H. ter Beek, A. Legay, and A. Lluch Lafuente. 2018. QFLan: A Tool for the Quantitative Analysis of Highly Reconfigurable Systems. In *FM'18 (LNCS, Vol. 10951)*. Springer, 329–337. https://doi.org/10.1007/978-3-319-95582-7_19