FTS4VMC: a front-end tool for static analysis and family-based model checking of FTSs with VMC

Maurice H. ter Beek[a,*], Ferruccio Damiani[b], Michael Lienhardt[c], Franco Mazzanti[a], Luca Paolini[b], Giordano Scarso[b]

[a] *ISTI–CNR, Pisa, Italy*
[b] *University of Turin, Turin, Italy*
[c] *ONERA, Palaiseau, France*

Abstract

FTS4VMC is a publicly available front-end tool for the static analysis and family-based model checking of a Featured Transition System (FTS). It can detect ambiguities in an FTS, disambiguate an ambiguous FTS, transform an FTS into a Modal Transition System (MTS), and interact with the VMC model checker for family-based verification.

Keywords: product families, variability, static analysis, model checking, VMC, featured transition systems, modal transition systems

Introduction

A Featured Transition System (FTS) is a formalism for capturing variability in behavioural models of product families or configurable systems [1, 2]. The behaviour of all variants (products) is modelled 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. As first recognised in the seminal paper [3], efficient model checking of FTSs is challenging, as the number of possible variants may be exponential in the number of features and each variant may moreover exhibit a large state space. Ideally, the compactness of an FTS is exploited to reason on the whole system at once. Such an *all-in-one*

*Corresponding author
Email addresses: maurice.terbeek@isti.cnr.it (Maurice H. ter Beek), ferruccio.damiani@unito.it (Ferruccio Damiani), michael.lienhardt@onera.fr (Michael Lienhardt), franco.mazzanti@isti.cnr.it (Franco Mazzanti), luca.paolini@unito.it (Luca Paolini), giordano.scarso@edu.unito.it (Giordano Scarso)

technique, by which the behaviour of all variants is examined only once simultaneously, is called family-based analysis; in contrast, in an enumerative product-based analysis, the behaviour of each individual variant is examined *one-by-one* [4]. During the past decade, FTSs were shown to be amenable to family-based model-checking [1, 2, 5–11].

In this paper, we present the front-end tool FTS4VMC for the research tool VMC [12–14], developed to make VMC amenable also to FTSs. VMC is a research tool for the analysis of a behavioural variability model of a product family or configurable system in its early design phase. It accepts as input a Modal Transition System with a set of logical *variability constraints* (MTSυ), akin to an FTS' feature expressions. The concept of a Modal Transition System (MTS) [15] was originally introduced in [16] to capture the refinement of partial descriptions into more detailed ones. MTSυs were introduced in [14] to compactly model product family behaviour, whose individual variant (product) behaviour can be obtained by means of a special-purpose refinement relation or by an equivalent operational derivation procedure. In [17], it was shown that such MTSs are equally expressive as FTSs.

We initiated the static analysis of FTSs in [18][1]. We identified and defined three ambiguities for an FTS: a *dead transition* (i.e., a transition that is unreachable, and thus cannot be executed, in any variant); a *false optional transition* (i.e., a transition that can be executed in all variants in which its source state is reachable); and a *hidden deadlock state* (i.e., a state from which a transition can be executed only in some variants). We presented an algorithm to detect such ambiguities in FTSs, with a correctness proof, and a procedure to remove them, mimicking the well-established anomaly detection for feature models [19]. The motivations were twofold: an ambiguous FTS is often undesired, since it gives an unclear idea of the SPL behaviour, and an unambiguous FTS paves the way for an efficient kind of family-based model checking[2]. We illustrated this in [18] on a few examples from the literature.

To improve the practical applicability of the automated static analysis for ambiguity detection in FTSs, in [20] we presented a new and more efficient algorithm by formalising the ambiguity criteria as propositional formulae, thus reducing ambiguity detection to SAT solving, with a correctness proof. We applied the new algorithm to a large set of benchmark examples from the literature, including the FTS of the complete mine pump model of [21] and

[1]This paper received the SPLC 2019 Best Paper Award and the ACM Badge "Artifacts Evaluated & Reusable".

[2]As explained in detail in the next section on Impact, an unambiguous FTS is such that any property ϕ specified in a rich fragment of a dedicated variability-aware temporal logic can be verified with a linear complexity and if ϕ is true, then ϕ is true in all variants.

that of the Claroline SPL of [22] with over 10,000 transitions, both of which were not tractable with the algorithm presented in [18]. Thus, we empirically demonstrated the improved efficiency of the new algorithm by means of a significant runtime speedup (8.5x on average for the FTS benchmarks that were tractable also before [20, Table 2]). A Python implementation of this algorithm can be found in `analyser.py`, which is freely available online from [23].

The front-end tool FTS4VMC was first presented during the conference SPLC 2021 in the Demonstrations and Tools track as a short paper [24] and as part of a tutorial on a toolchain involving FTS4VMC (reported in [25]). The current paper is an extended version of [24], focussing on the software.

The toolchain constituted by FTS4VMC, `analyser.py` from [23] (which implements the static analysis algorithm from [20], where it was shown to be more efficient than the one presented in [18]), and VMC can be used to i) analyse an FTS for ambiguities, ii) remove ambiguities from an FTS, and iii) perform an efficient kind of family-based verification of properties of an FTS without hidden deadlock states (for which the FTS is transformed into an MTS). This is possible through a user-friendly GUI, which implements the code that allows to upload and download files, handle users' session data, render graph visualisation and HTML output, and communicate with VMC. Moreover, the GUI predisposes the seamless interaction with the following main Python procedures: `analyser.py`, 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. Alternatively, the core classes of FTS4VMC can be used from a command-line interface, which is useful for particularly large FTSs that cannot easily be rendered in a visually attractive format.

Impact

FTS4VMC automates the engineering and verification methodology envisioned in Fig. 1, which offers a user the possibility to verify properties of an FTS or an MTSv, as well as the possibility to improve an FTS by removing ambiguities[3]. The verification strategy sketched in Fig. 1, which will be

[3]A dead transition indicates a modelling error that must be corrected. A false optional transition indicates a model redundancy that may be intentional, but resolving it allows for more efficient verification options. A hidden deadlock should be made explicit to improve

discussed in more detail in the next section, is as follows. If i) the model is live[4], 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$^{\Box}$[5], then ϕ can be verified directly on the underlying MTS (ignoring the feature expressions in case of an FTS model) with a linear complexity. Moreover, if iii) ϕ holds, then this validity is preserved in all variants, i.e. ϕ holds for all variants. However, if any of these three conditions does not hold, then ϕ has to be verified externally with classical family-based model-checking tools (in case of an FTS model) or with VMC through product-based model checking (in case of an MTSυ model), with an exponential complexity [1, 7].

Conclusion

FTS4VMC expands the possible use cases of the VMC model checker for behavioural variability models to include also FTSs, offering moreover a significant gain in efficiency in specific cases, as demonstrated in [20].

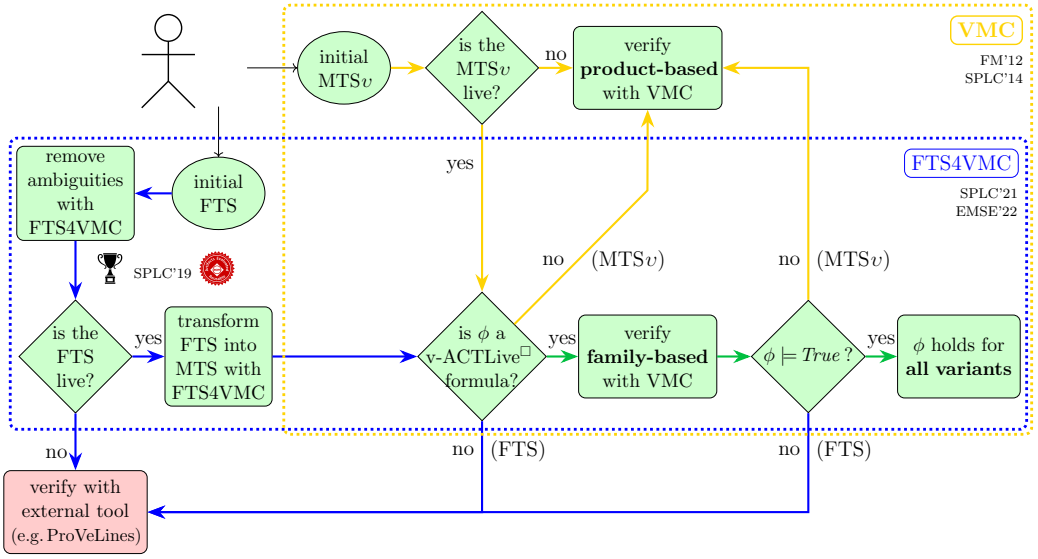

Figure 1: Engineering and verification methodology from [24]

model understanding and to enable an efficient kind of family-based verification (if the deadlocks in the variants that are the cause should not be remedied in the first place).

[4]An FTS (or MTSυ) is said to be *live* if all states are live, where a live state is such that it does not occur as a deadlock state (from where no action is possible) in any variant.

[5]v-ACTL [14, 26] is a variability-aware action-based and state-based branching-time temporal logic derived from ACTL (an action-based version of the well-known logic CTL) and v-ACTLive$^{\Box}$ is a rich fragment of v-ACTL interpreted on live MTSs.

Software

FST4VMC was developed using JavaScript coupled with a small back-end based on the Flask micro web framework. Flask was preferred over similar alternatives because it is simple to deploy locally as a desktop application and it provided a direct integration with the previously developed code (i.e., `analyser.py` [23]). Nonetheless, the online repository[6] makes the core classes of FTS4VMC (viz., `Disambiguator`, to automatically remove ambiguities detected by the analyser, and `Translator`, to transform FTSs into MTSs) available as two separate programs (viz., `disambiguate.py` and `translate.py`) that allow to use FTS4VMC from a command-line interface. This is particularly useful for large FTSs that cannot easily be rendered in a visually attractive format.

Tool Functionality

FTS4VMC fully automates the following steps of the engineering and verification methodology envisioned in Fig. 1:

Input Specify or upload an FTS expressed as a directed graph in Graphviz's `dot` file format[7], including a definition of the feature model using the attribute `FM`[8]. Nodes represent states and edges represent transitions. Every edge requires an action and feature expression defined inside the label `attribute` and exactly one node must have the attribute `initial` set to *True*. We refer to [20] for definitions of an FTS, a feature model, and a feature expression as used in FTS4VMC, while we refer to the folder tests/dot [9] in the online repository for a dozen exemplary FTS models from the literature that can be uploaded.

Ambiguity detection Check if the FTS is ambiguous[10], by calling method `analyser.z3_analyse_full` of `analyser.py`, and output an updated version of the FTS by calling `Disambiguator.highlight_ambiguities` to change the `dot` file of the FTS to highlight the detected ambiguities: dead transitions are highlighted in blue, false optional transitions in

[6] `https://github.com/fts4vmc/FTS4VMC`

[7] This is a well-known graphical notation supported by the Graphviz open-source graph visualisation software (cf. `https://www.graphviz.org`).

[8] This is a propositional logic formula that defines which feature configurations represent valid variants, where each feature is *True* if it is selected and *False* if unselected.

[9] `https://github.com/fts4vmc/FTS4VMC/tree/master/tests/dot`

[10] An FTS is said to be ambiguous if at least one of its states is a hidden deadlock or at least one of its transitions is dead or false optional.

green, and hidden deadlock states in red (included in the graph rendering with a legend reporting the three types of ambiguities, cf. Fig. 2).

Ambiguity removal Remove all or some selected types of ambiguities from an FTS, by calling the `solve_hidden_deadlocks`, `remove_transitions` and `set_true_list` (set feature expressions of false optional transitions to *True*, without removing them) methods of `disambiguator.py`. We refer to [20, Section 4] for definitions of the three types of ambiguities that FTS4VMC can detect and of how to remove them from an FTS.

Liveness analysis Check if the FTS is live, i.e., if it has hidden deadlock states (ignoring the detection of dead and false optional transitions), by calling method `analyser.z3_analyse_hdead` of `analyser.py`. This is a specialised implementation of the static analysis algorithm that represents a hidden deadlocks discovery algorithm (i.e., analysing only liveness) [20, Section 5.3], which requires only a fraction of the runtime of the full ambiguities discovery algorithm (4.2% on average for the three largest FTS benchmarks [20, Section 6.2], i.e. those that were not tractable with the algorithm from [18]).

Transformation Transform a (live) FTS into an MTS, by calling first the `Translator.load_model` method and then the `Translator.translate` method of `translator.py`. Given the FTS representation in `dot` format, the translation process creates an assignment for every state with at least one outgoing transition. Each assignment contains a list of statements for every outgoing transition defined as follows: the action label of the transition, the transition's optionality inside parentheses, and the destination state of the transition preceded by a dot (`.`). A final step adds the statements `SYS` and `Constraints { LIVE }` to specify which state is the initial state and to declare that the MTS is live, respectively, as required by VMC's syntax of MTS*υ*s. Method `get_mts` of `graph.py` allows to display the FTS's corresponding MTS at any time. This transformation makes use of the fact that we know from [20] that any FTS $\mathcal{F}$ can be transformed into an MTS $\mathcal{F}_{\text{MTS}}$ in a straightforward manner such that whenever $\mathcal{F}$ is live, then also $\mathcal{F}_{\text{MTS}}$ is live.

Property verification Specify a v-ACTL formula in FTS4VMC and verify it with VMC, by calling `vmc_controller.py`, after which the output of VMC is displayed. If the formula is a v-ACTLive$^{\Box}$ formula and it holds, then FTS4VMC reports that it holds for all variants of the FTS. Instead, if the formula is not a v-ACTLive$^{\Box}$ formula or it does not hold, then FTS4VMC reports that the result of the formula is not necessarily

preserved by the variants (i.e., the user needs to resort to product-based model checking with VMC). If the formula is false, FTS4VMC also offers to display the counterexample provided by VMC by calling `VmcController.run_vmc` and its getter methods. This kind of family-based verification of properties of (live) FTSs relies on the fact that a result for MTSυs, namely for a property expressed in v-ACTLive$^{\Box}$, validity for the family model guarantees validity of the property for all product models (cf. [14, Theorem 4]), carries over to live FTSs [20].

The novelties introduced in [24] are clearly indicated in Fig. 1: the blue and the green steps (arrows) if applied to FTSs are made possible by FTS4VMC.

Tool Views

During execution of the above steps of the tool's functionality, the user can always stop the processing and switch between different views (cf. Fig. 2):

Console Displays progress and the results of the performed steps.

Source Displays the `dot` source file of the current FTS or MTS.

Graph Displays the rendered graph in `svg` format, highlighting the feature model and ambiguities, by using the `pydot` library[11].

Summary Displays the console output after successful analysis in a visually attractive format.

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` format and with or without highlighted ambiguities, the transformed MTS, etc.).

Reproducibility

The specifications of all the FTS models needed to reproduce the benchmark experiments presented in [20, §6 and §7] are freely available online [23].

[11] By default only models with at most 300 transitions are rendered. This limit is defined in `config.py` and can be changed if desired. Moreover, by default graphs are rendered with a top to bottom (`TB`) layout for better readability on a Web browser, but also this can also be configured by changing the value of `config.RENDER_GRAPH_DIRECTION`.

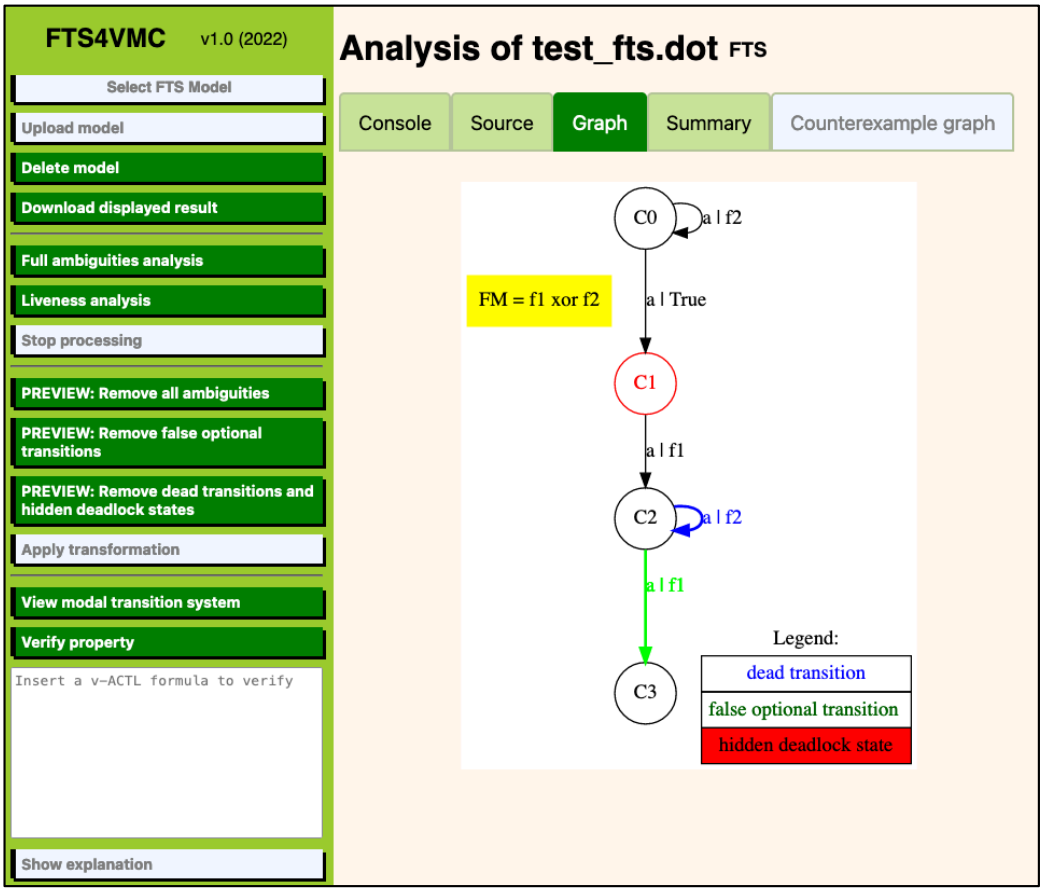

Figure 2: Screenshot of FTS4VMC's GUI

Documentation

Documentation of the software is available on GitHub[12], explaining how to install and run FTS4VMC or to use its core classes from a command-line interface. The GitHub page also provides information on the internal implementation (i.e., the structure of the source code), which is important to allow other developers to understand the source code and possibly modify it to their needs, as well as a link to an online user guide[13].

[12] `https://github.com/fts4vmc/FTS4VMC#readme`
[13] `https://github.com/fts4vmc/FTS4VMC/blob/master/MANUAL.md`

Acknowledgements

Maurice ter Beek and Luca Paolini acknowledge funding from the Italian MIUR-PRIN 2020TL3X8X project T-LADIES (Typeful Language Adaptation for Dynamic, Interacting and Evolving Systems).

References

[1] A. Classen, M. Cordy, P. Schobbens, P. Heymans, A. Legay, J. Raskin, 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. doi:10.1109/TSE.2012.86.

[2] M. Cordy, X. Devroey, A. Legay, G. Perrouin, A. Classen, P. Heymans, P. Schobbens, J. Raskin, A Decade of Featured Transition Systems, in: M. H. ter Beek, A. Fantechi, L. Semini (Eds.), From Software Engineering to Formal Methods and Tools, and Back, Vol. 11865 of LNCS, Springer, 2019, pp. 285–312. doi:10.1007/978-3-030-30985-5_18.

[3] A. Classen, P. Heymans, P. Schobbens, A. Legay, J. Raskin, Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines, in: Proceedings of the 32nd International Conference on Software Engineering (ICSE'10), ACM, 2010, pp. 335–344. doi:10.1145/1806799.1806850.

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

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

[6] M. Cordy, A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, ProVeLines: A Product Line of Verifiers for Software Product Lines, in: Proceedings of the 17th International Software Product Line Conference (SPLC'13), Vol. 2, ACM, 2013, pp. 141–146. doi:10.1145/2499777.2499781.

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

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

[9] M. H. ter Beek, E. P. de Vink, T. A. C. Willemse, Family-Based Model Checking with mCRL2, in: M. Huisman, J. Rubin (Eds.), Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering (FASE'17), Vol. 10202 of LNCS, Springer, 2017, pp. 387–405. doi:10.1007/978-3-662-54494-5_23.

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

[11] M. H. ter Beek, S. van Loo, E. P. de Vink, T. A. Willemse, Family-Based SPL Model Checking Using Parity Games with Variability, in: H. Wehrheim, J. Cabot (Eds.), Proceedings of the 23rd International Conference on Fundamental Approaches to Software Engineering (FASE'20), Vol. 12076 of LNCS, Springer, 2020, pp. 245–265. doi:10.1007/978-3-030-45234-6_12.

[12] M. H. ter Beek, F. Mazzanti, A. Sulova, VMC: A Tool for Product Variability Analysis, in: D. Giannakopoulou, D. Méry (Eds.), Proceedings of the 18th International Symposium on Formal Methods (FM'12), Vol. 7436 of LNCS, Springer, 2012, pp. 450–454. doi:10.1007/978-3-642-32759-9_36.

[13] M. H. ter Beek, F. Mazzanti, VMC: Recent Advances and Challenges Ahead, in: Proceedings of the 18th International Software Product Line Conference (SPLC'14), Vol. 2, ACM, 2014, pp. 70–77. doi:10.1145/2647908.2655969.

[14] M. H. ter Beek, A. Fantechi, S. Gnesi, F. Mazzanti, 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. doi:10.1016/j.jlamp.2015.11.006.

[15] J. Křetínský, 30 Years of Modal Transition Systems: Survey of Extensions and Analysis, in: L. Aceto, G. Bacci, G. Bacci, A. Ingólfsdóttir, A. Legay, R. Mardare (Eds.), Models, Algorithms, Logics and Tools, Vol. 10460 of LNCS, Springer, 2017, pp. 36–74. doi:10.1007/978-3-319-63121-9_3.

[16] K. G. Larsen, B. Thomsen, A Modal Process Logic, in: Proceedings of the 3rd Symposium on Logic in Computer Science (LICS'88), IEEE, 1988, pp. 203–210. doi:10.1109/LICS.1988.5119.

[17] M. H. ter Beek, F. Damiani, S. Gnesi, F. Mazzanti, L. Paolini, On the expressiveness of modal transition systems with variability constraints, Sci. Comput. Program. 169 (2019) 1–17. doi:10.1016/j.scico.2018.09.006.

[18] M. H. ter Beek, F. Damiani, M. Lienhardt, F. Mazzanti, L. Paolini, Static Analysis of Featured Transition Systems, in: Proceedings of the 23rd International Systems and Software Product Line Conference (SPLC'19), ACM, 2019, pp. 39–51. doi:10.1145/3336294.3336295.

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

[20] M. H. ter Beek, F. Damiani, M. Lienhardt, F. Mazzanti, L. Paolini, Efficient static analysis and verification of featured transition systems, Empir. Softw. Eng. 22 (1) (2022) 10:1–10:43. doi:10.1007/s10664-020-09930-8.

[21] A. Classen, Modelling and model checking variability-intensive systems, Ph.D. thesis, University of Namur (2011).

[22] X. Devroey, G. Perrouin, M. Cordy, P. Schobbens, A. Legay, P. Heymans, Towards Statistical Prioritization for Software Product Lines Testing, in: Proceedings of the 8th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'14), ACM, 2014, pp. 10:1–10:7. doi:10.1145/2556624.2556635.

[23] M. H. ter Beek, F. Damiani, M. Lienhardt, F. Mazzanti, L. Paolini, Supplementary material for: "Static Analysis of Featured Transition Systems" (December 2019). doi:10.5281/zenodo.2616646.

[24] M. H. ter Beek, F. Mazzanti, F. Damiani, L. Paolini, G. Scarso, M. Valfrè, M. Lienhardt, Static Analysis and Family-based Model Checking of Featured Transition Systems with VMC, in: Proceedings of the 25th International Systems and Software Product Line Conference (SPLC'21), Vol. 2, ACM, 2021, pp. 24–27. doi:10.1145/3461002.3473071.

[25] M. H. ter Beek, F. Mazzanti, F. Damiani, L. Paolini, G. Scarso, M. Lienhardt, Static Analysis and Family-based Model Checking with VMC, in: Proceedings of the 25th International Systems and Software Product Line Conference (SPLC'21), Vol. 1, ACM, 2021, p. 214. doi:10.1145/3461001.3472732.

[26] M. H. ter Beek, F. Damiani, S. Gnesi, F. Mazzanti, L. Paolini, From Featured Transition Systems to Modal Transition Systems with Variability Constraints, in: R. Calinescu, B. Rumpe (Eds.), Proceedings of the 13th International Conference on Software Engineering and Formal Methods (SEFM'15), Vol. 9276 of LNCS, Springer, 2015, pp. 344–359. doi:10.1007/978-3-319-22969-0_24.

Required Metadata

Current code version

Nr.	Code metadata description	Please fill in this column
C1	Current code version	v1.0.0
C2	Permanent link to code/repository used for this code version	https://github.com/fts4vmc/FTS4VMC
C3	Permanent link to Reproducible Capsule	https://doi.org/10.24433/CO.8774017.v1
C4	Legal Code License	GPL-3.0
C5	Code versioning system used	Git
C6	Software code languages, tools, and services used	Graphviz, JavaScript, Python 3, Z3
C7	Compilation requirements, operating environments & dependencies	Flask, lrparsing, pip, puremagic, pydot, z3-solver
C8	If available Link to developer documentation/manual	https://github.com/fts4vmc/FTS4VMC/blob/master/README.md
C9	Support email for questions	fts4vmc@di.unito.it

Table 1: Code metadata

Current executable software version

Nr.	(Executable) software metadata description	Please fill in this column
S1	Current software version	v1.0.0
S2	Permanent link to executables of this version	`https://github.com/fts4vmc/FTS4VMC`
S3	Permanent link to Reproducible Capsule	`https://doi.org/10.24433/CO.8774017.v1`
S4	Legal Software License	GPL-3.0
S5	Computing platforms/Operating Systems	Debian GNU/Linux, macOS, Windows
S6	Installation requirements & dependencies	Python 3, Flask, lrparsing, pip, puremagic, pydot, Graphviz dot, z3-solver
S7	If available, link to user manual - if formally published include a reference to the publication in the reference list	`https://github.com/fts4vmc/FTS4VMC/blob/master/README.md` `https://github.com/fts4vmc/FTS4VMC/blob/master/MANUAL.md`
S8	Support email for questions	fts4vmc@di.unito.it

Table 2: Software metadata