Daedalux: An Extensible Platform for Variability-Aware Model Checking

Sami Lazreg, Maxime Cordy, Simon Thrane Hansen
{sami.lazreg,maxime.cordy,simon.hansen}@uni.lu
SnT, University of Luxembourg
Luxembourg

Axel Legay
axel.legay@uclouvain.be
Université Catholique de Louvain
Belgium

ABSTRACT

This paper presents Daedalux, a new model-checking platform for variability-intensive systems based on *Featured Transition System* theory developed in C++. Daedalux features a modular, flexible, and extensible architecture, overcoming previous tools' maintainability limitations. In addition, during verification, it provides visualizations of intermediate models and results. A key added value of Daedalux lies in its software architecture, which allows straightforward extension and integration of new formalisms and verification algorithms. We have implemented two recent FTS-based approaches, i.e., a statistical model-checking algorithm for LTL properties and an exhaustive algorithm for multi-LTL properties. By reducing the entry barrier of understanding variability-aware model checking and facilitating the comprehension and extension of the software tools, we hope to increase the community's ambitions in developing novel model-checking advances. A video demonstration of Daedalux can be found at https://youtu.be/kirpOAlV-0w.

ACM Reference Format:
Sami Lazreg, Maxime Cordy, Simon Thrane Hansen and Axel Legay. 2024. Daedalux: An Extensible Platform for Variability-Aware Model Checking. In *2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3639478.3640043

1 INTRODUCTION

Variability plays an essential role in developing modern systems to cope with the increasing demand for customization and to address different requirements. Software Product Lines (SPLs) [28] is an engineering paradigm that has emerged to efficiently develop and maintain a *family* of similar software products called *variants*. Each variant is a specific SPL configuration, defined by a distinct set of variation points called *features*. A *feature model* (FM) is a tree-like structure that represents the set of features of an SPL and their relationships to denote the valid variants of the SPL.

While SPLs are a promising approach to reduce development costs and time-to-market, they introduce new challenges in terms of verification and validation, as the number of variants can be exponential in the number of features. In particular, an engineer must ensure that all the variants they build satisfy the intended requirements. Model checking [2] is a formal verification technique that enables automatically verifying the correctness of a system expressed as a model concerning a given property expressed as a temporal logic formula (e.g., LTL [27]). Unfortunately, the traditional model checking approach is unsuitable for SPLs because it can only verify each of the (numerous) variants separately.

Featured Transition Systems (FTS) [7, 9] are a variability-aware representation of an SPL that extends standard Transition Systems (TS) with an FM by annotating each transition with a *feature expression* – a symbolic encoding of the set of variants able to execute the transition. E.g., Listing 2 (in Sec. 3) models an FTS of a motor, such that the *Alarm* feature can force the motor to stop in case of danger. Thus, a single FTS can represent the behaviour of all SPL variants, so specialized – *variability-aware* – algorithms can exploit the common parts of the FTS to reduce the verification time of all variants [6, 7].

FTSs have, since their introduction, been extended in various directions to increase both their formalism expressiveness and their verification capabilities [9]. However, many challenges remain to be addressed to enable their broad application. For example, extending FTSs to verify cyber-physical systems [9, 23] necessitates advancements in formalism expressiveness and verification algorithms.

All these past and future extensions raise the necessity for a sustainable software platform to support the development of FTS-based model checking algorithms and tools. Unfortunately, the existing tools – SNIP [5] and ProVeLines [8] – suffer from maintainability issues. SNIP cannot accommodate the incorporation of new extensions, whereas ProVeLines employs *#ifdef* macros to manage code fragments associated with specific extensions, thereby engendering unwieldy, macro-filled monolithic code blocks.

To facilitate the development of such methods and tools, we propose *Daedalux*[1], a flexible and extensible platform to implement and integrate FTS-related formalisms and algorithms. In addition to suitable design structures to implement model-checking algorithms, Daedalux provides a set of visualizations to facilitate the comprehension of intermediate models and results. We have used Daedalux ourselves to implement three classes of verification procedures: the original, exhaustive FTS model-checking algorithm [6] for linear-time properties, an alternative based on Statistical Model Checking (SMC) [11] and an exhaustive algorithm to check FTS against multi-properties [13]. The successful implementation of these three different verification approaches within the same codebase demonstrates the success of Daedalux in providing a flexible

[1] https://github.com/samilazregsuidi/Daedalux

ICSE-Companion '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0502-1/24/04.
https://doi.org/10.1145/3639478.3640043

platform for developers of model checking tools. The underlying approaches can also be straightforwardly combined, e.g., we get "for free" a *statistical* algorithm to check FTSs against multi-properties. With Daedalux, we hope to ignite the software verification research community in developing novel, variability-aware model-checking approaches of increasing complexity by reducing the implementation burden.

2 RELATED WORK

SNIP [5] and ProVeLines [8] are tools that verify all products at once directly in the checking engine. SNIP was the first tool to support the verification of SPLs using FTSs and Linear Temporal Logic as specifications. It was later extended to ProVeLines to support more features and verification algorithms. In parallel, Ter Beek et al. developed mCRL2 [31], a model checker capable of verifying the full spectrum of mu-calculus properties over non-quantitative FTS in a family-based context. Compared to these, Daedalux provides a more flexible and extensible architecture to facilitate the integration of new formalism such as hyper properties [13] and verification algorithms such as a family-based extension of SMC [11, 26].

Other tools can indirectly verify SPLs by reducing the problem to traditional model checking. ProFeat [3] translates a stochastic SPL into a regular probabilistic model checking problem [15] manageable by the PRISM model checker [24]. Another approach by Ter Beek [30] represents non-stochastic SPLs as modal transition systems with may and must features. Dimovski et al. [12, 14] developed a 3-value game framework to lift FTS into a TS verified with SPIN [20]. This approach is based on a 3-valued game framework and has later been extended to the full mu-calculus [14]. Some tools allow us to specify incremental and dynamic feature extensions. Another example is the QFLAN approach [33], which builds upon SMC to analyze highly reconfigurable systems with featured constraints at runtime. These approaches exhibit theoretical shortcomings and do not harness the capabilities inherent to variability-aware tools.

3 DAEDALUX EXAMPLE AND FRAMEWORK

Example. Daedalux takes as inputs 1) a feature model [22] describing the SPL features; 2) an FTS model specifying the behavior of the SPL variants; 3) a property to check. Currently, Daedalux supports feature models written in *Text Variability Language (TVL)* [4] (see Listing 1) and FTS written in *fPromela* (see Listing 2), a feature-aware and stochastic extension of SPIN's Promela language [20] that allows using featured expressions. Properties are commonly expressed as temporal logic formulas; Daedalux currently supports LTL [27] and its extension Multi-LTL [18].

Listing 1: Excerpt of the Motor FTS written in fPromela

```
active proctype Motor() {
    do :: safe ->
       if :: skip;
          :: safe = false; danger = true fi;
       :: danger ->
       gd :: Alarm -> danger = false; stopped = true;
          :: else -> skip; dg;
       :: stopped ->
       if :: skip;
          :: stopped = false; safe = true;
       fi;
    od; }
```

Listing 2: Motor feature model in TVL

```
root Motor group allOf { opt Alarm }
```

Usage. Users can use a Command-Line Interface (CLI) to verify the motor SPL modeled in fPromela and TVL against a specified property. Specifically, the following command: `$./daedalux check --exhaustive -ltl='[](danger-><>stopped)' motor.pml` checks that the motor variants stop when danger occurs. Running the command reveals that only the variant with feature *Alarm* satisfies the property. A counter-example (trace of model execution) is provided for the other variant (without feature *Alarm*).

Framework. The process of the Daedalux framework is portrayed in Fig. 1. The first step is to parse the fPromela and TVL models. The *parsing stage* transforms the input into abstract syntactic models (i.e., *AST*, *Symbol Table* and *Program Graph Automata*), which are subsequently utilized by the *semantic engine* to construct the TS, including the initial state, the successors of a state (denoted as $Post(s)$), executable transitions, and more. The internal structure of the states, i.e., features, stochasticity, and composition, are encapsulated by an API utilized by the *verification algorithm* to build, explore, and check the TS to output the lists of valid and invalid products with their execution traces as counterexamples. Daedalux can output models both in textual and visual formats. Daedalux can verify LTL properties on stochastic and non-stochastic SPLs using dedicated (exhaustive or statistical) model checking algorithms [26]. Furthermore, it supports multi-LTL by extending the application of the SMC algorithm to multi-FTS [13].

4 DAEDALUX ARCHITECTURE

Fig. 2 illustrates the five primary modules forming the architecture of *Daedalux*. The *Symbol* module provides a meta-model to support symbol management, whereas the *AST* module allows representing the formal model in an AST. The module *Automata* provides a program graph meta-model to enrich the AST with control flow through nodes and edges. The *Semantic* module implements the basic notions of the *FTS*, such as features, states, and transitions, that the verification algorithm in the *Algorithm* module works on.

Understandability. We propose to reuse architectural and design patterns in the different modules to facilitate future extensions, such as new formalisms and reasoning techniques. For example, the proposed *AST* module is based on the traditional composite design pattern [16], in which a node can be either an expression or a statement. The symbol table maintains a collection of symbols, where each symbol can signify either a variable definition (*VarDef*) or a new variable type (*TypeDef*), encompassing various entities like data structures, threads, functions, and more. The module *Automata* follows the same philosophy.

We propose a new flexible meta-model for the *Semantic* module to represent *TSs*. A variable has a datatype and a value that may change over time. The datatype defines the variable's domain and memory representation. We consider two categories of data types: scalar and compound. Scalars are primitives or pointers, while compound types encompass arrays and more intricate data structures. In this context, we represent the state of a TS as an executable compound variable, exhibiting a value and a list of executable transitions. Applying (or firing) a transition to the state may modify its instruction pointer (location in the program graph) and the values

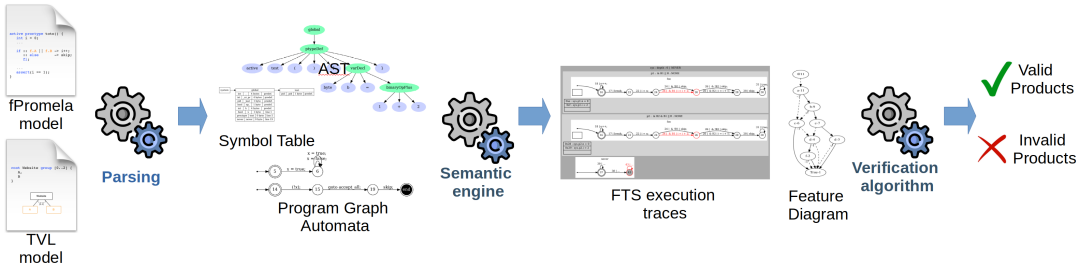

Figure 1: Daedalux Framework Illustration.

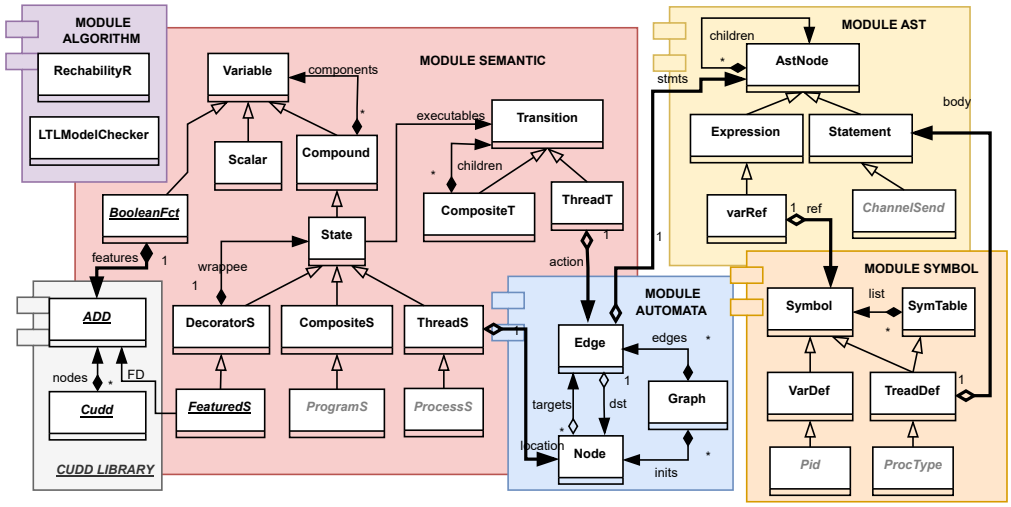

Figure 2: Excerpt of Daedalux Software Architecture.

of its variables. Using the composite design pattern, a state can be a parallel or synchronous composition of states (see Fig. 2).

Abstraction. Daedalux proposes two fundamental abstractions. The first one aims to decouple the formalisms from their verification algorithms to separate the syntactic and the semantics modules, as demonstrated by the FTS. The representation of an FTS is abstracted through generic *Symbol*, *AST* and *Automata* modules, while its verification is abstracted through the *Semantic* and *Algorithms* modules. This decoupling enables one to explore various verification algorithms of the same FTS in the same run.

The second abstraction focuses on the fundamental concepts of TSs. Each state implements the successor relation $Post : S \rightarrow S^n$ and indicates whether or not it is an accepting state $Accept : S \rightarrow \mathbb{B}$. As a result, the same model-checking algorithm can be applied on *FTS* or *Multi-FTS* [13], despite variations in their underlying state structures. For example, an *FTS* state $s_{FTS} \in S_{FTS}$ is a parallel composition of threads (processes in *Promela* semantic) such as $s_{FTS} = \{p_0 + \ldots + p_n\}$ while an *Multi-FTS* state [13] is a synchronous composition of *FTS* states such as $s_{M_{FTS}} = \{s_{FTS_0} \times \ldots \times s_{FTS_n}\}$.

Our current *LTL* model-checking algorithm follows the well-established approach of automata-based model checking. Thanks to our abstraction, the algorithm seamlessly accommodates both $LTL_{mc}(FTS \times BA_{LTL})$ and $LTL_{mc}(M_{FTS} \times BA_{M_{LTL}})$, enabling a unified approach[2]. This versatility extends to our SMC algorithm, which equally supports *Bounded LTL*, *Bounded fMulti LTL* [13], and stochastic systems [26]. It should be noted that these abstractions allow the algorithm to accommodate various system types, including stochastic and deterministic, featured and single-system, composite, or unitary, among others.

Extensibility. Daedalux is designed as an extensible framework that leverages templates and base classes to offer fundamental functionalities for FTS model checking. The featured semantics have been implemented using a *Transition System* decorator. This decorator transforms a non-featured state into a featured state, thereby altering how it manages its set of executable transitions and the consequences of applying them. This is useful for adding featured semantics[3] to any state formalism such as *Markovian Decision Process* and *Modal Transition System* to name a few.

Another software design decision is to include the features of a state as a state variable. In this context, two featured states can exhibit various relationships: they can be equivalent, different, or one may imply the other, signifying that the feature expression of the first state generalizes the one of the second state, formally

[2]We utilize ltl2ba to transform LTL formula into Büchi automaton [17].
[3]The featured decorator constrains the possible executable transitions using its feature expression as a guard.

$(s, f), (s', f') \in S_f, s = s' \wedge [\![f]\!] \subset [\![f']\!]$. The featured extension comprises the classes in italics and underlined in Fig. 2. The CuDD library [29] is used as an *Algebraic Decision Diagram* (ADD) library. To support the Promela language, we provide an extension (i.e., the grey italics classes) that handles and manages Promela-specific symbols, statements, variables, and state semantics [20].

Modularity. We tried to reduce coupling between the different modules. As a result, we generally have only one dependency per module. For example, the *AST* only requires the *Symbol* module (i.e., to link the variable reference to the symbol data). Executable symbol definitions such as function and thread point to their statements in the *AST*. The module *Automata* transforms the AST into a program graph with control flow through nodes and edges. The edges are linked to the statement, representing its action. Thread transitions are linked to the edge to execute. The thread state has a location variable representing the thread location in the program graph. A featured state has a boolean function variable representing its features. This function encapsulates an *ADD* from the *CuDD* library and provides an interface to operate on the underlying *ADDs*.

The presented architecture aims to be extensible to easily integrate future developments such as i) adding a new input language, ii) adding a new model checking algorithm, or iii) a new formalism. To add a new input language for an existing formalism (for example, the Prism language for modelling stochastic FTS), the developer must only write a parser that translates the input language elements into an AST and the necessary symbol tables. Adding a new formalism, such as hybrid FTS, requires more work as it involves extending the *AST*, *Symbol*, and *Semantic* modules to capture all aspects of the new formalism. Specifically, classes in the *Symbol* module must be added for the new types and functions, while the new statements and expressions are captured by the changes to the *AST*. The semantic management of these new elements is done by extending the *Post* and *Apply* method of the *Thread* class[4]. The complexity of adding a new model-checking algorithm to an existing formalism depends on its interaction with the *State* class. If the algorithm only explores the state space through the *Post*, *Apply*, and *Accept* methods, then no modifications of the existing code are needed, as the existing (and generic) reachability relation class can be directly used. However, if the new algorithm requires exploring the sub-structure of the state, modification in the reachability class or *Semantic* module may be required.

5 PERSPECTIVES AND CONCLUSION

With Daedalux, we make an essential step in democratizing verification approaches for variability-intensive systems. The inherent need to handle system variability introduces a complexity that crosscuts all modelling formalisms and verification algorithms. While this necessitates the development of a dedicated tool equipped with variability management mechanisms, the parallel endeavor to accommodate and expand upon various model-checking techniques initially designed for single (non-variable) systems poses a challenge in software maintenance. It can result in mounting technical debt, hindering the implementation of innovative approaches within previously successful tools. Daedalux, with its robust architectural design, bridges this gap and paves the way for the continued growth of variability-aware model checking. It thereby expands its accessibility to a broader community, including those unfamiliar with this research field's complete historical evolution.

The democratization of SPL model checking, as bootstrapped by Daedalux, supports SPL engineers in verifying that *all* variants they build satisfy essential requirements. With our tool, engineers can explore the behaviour of large variant sets to answer questions such as "Which variants never reach an unsafe state?" and "Which variants guarantee the delivery of a specific service with a 99% service level agreement?". Overall, the goal of Daedalux is to enable practitioners to use model checking innovations in an SPL context, complementing traditional quality assurance (i.e., testing) with theoretically grounded, model-based approaches.

We see several avenues for future work. Regarding expressiveness, we plan to extend Daedalux to support several of the existing FTS formalisms, such as FTS with hybrid behaviour and/or dynamic variability. A set of tailored verification algorithms will support each formalism. Additionally, we intend to introduce well-established optimizations, such as symbolic model checking [6] and experiment further with SMC to bolster the verification of large and intricate (hybrid) FTS [11, 32].

Exploring additional research directions, we consider integrating the *QFLAN* approach into our SMC technique. This integration would enable us to address quantitative logic that incorporates algebraic assumptions about system features, allowing for actions to be restricted when certain constraints are exceeded in relation to a predefined budget within a family-based context. Another possibility is to expand the work on hyperlogic [13] by considering non-deterministic and timing/probabilistic aspects simultaneously, as is the case for simple products [1, 25]. This would allow for modelling both the attacker's environment and the internal system response to various variations. We could then deduce which variants are the most robust against a specific attack.

We also want to expand our approach with counterexample-guided abstraction refinement (CEGAR) techniques [10]. In particular, in cases where the system does not satisfy the property, fraudulent simulations could be used to guide the engineer to correct the SPL. Furthermore, by leveraging synthesis techniques from the field of statistics and CEGAR-based approaches, automatic corrections could be proposed[19].

Usability is a crucial aspect of any software tool; we aim to improve the usability of Daedalux by extending the input formalisms to support more languages, such as Prism [24], QFLAN, or CLAFER [21]. We also plan to improve the user experience by adding a GUI to facilitate adoption by non-experts and practitioners. In this context, the main challenge will be to find an intuitive and visually useful graphical representation for large-scale systems. Another challenge will be to define an approach for visualizing errors and making them understandable from the description language.

6 ACKNOWLEDGEMENT

The contribution of Sami Lazreg, Maxime Cordy, and Simon Thrane Hansen to this work was funded by FNR Luxembourg (grant C19/IS/13566661/BEEHIVE/Cordy and grant INTER/FNRS/20/15077233/Scaling Up Variability/Cordy). The contribution of Axel Legay to this work was funded by FNRS Belgium (grant PDR T.0137.21).

[4]These codes are encapsulated using the Visitor design pattern to facilitate extension.

REFERENCES

[1] Erika Ábrahám, Ezio Bartocci, Borzoo Bonakdarpour, and Oyendrila Dobe. 2020. Probabilistic Hyperproperties with Nondeterminism. In *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12302)*, Dang Van Hung and Oleg Sokolsky (Eds.). Springer, 518–534. https://doi.org/10.1007/978-3-030-59152-6_29

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

[3] Philipp Chrszon, Clemens Dubslaff, Sascha Klüppelholz, and Christel Baier. 2018. ProFeat: feature-oriented engineering for family-based probabilistic model checking. *Formal Aspects Comput.* 30, 1 (2018), 45–75.

[4] Andreas Classen, Quentin Boucher, and Patrick Heymans. 2011. A text-based approach to feature modelling: Syntax and semantics of TVL. *SCP* 76 (December 2011), 1130–1143. Issue 12.

[5] Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. 2012. Model checking software product lines with SNIP. *Int. J. Softw. Tools Technol. Transf.* 14, 5 (2012), 589–612.

[6] 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.

[7] 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 *ICSE 2010, Proceedings*, Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel (Eds.). ACM, 335–344.

[8] Maxime Cordy, Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. 2013. ProVeLines: a product line of verifiers for software product lines. In *SPLC*. ACM, New York, NY, USA, 141–146.

[9] Maxime Cordy, Xavier Devroey, Axel Legay, Gilles Perrouin, Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Jean-François Raskin. 2019. A Decade of Featured Transition Systems. In *From Software Engineering to Formal Methods and Tools, and Back - Essays Dedicated to Stefania Gnesi on the Occasion of Her 65th Birthday (LNCS, Vol. 11865)*, Maurice H. ter Beek, Alessandro Fantechi, and Laura Semini (Eds.). Springer, 285–312.

[10] Maxime Cordy, Patrick Heymans, Axel Legay, Pierre-Yves Schobbens, Bruno Dawagne, and Martin Leucker. 2016. Counterexample guided abstraction refinement of product-line behavioural models. In *Software Engineering 2016, Fachtagung des GI-Fachbereichs Softwaretechnik, 23.-26. Februar 2016, Wien, Österreich (LNI, Vol. P-252)*, Jens Knoop and Uwe Zdun (Eds.). GI, 79–80. https://dl.gi.de/handle/20.500.12116/732

[11] Maxime Cordy, Sami Lazreg, Mike Papadakis, and Axel Legay. 2021. Statistical model checking for variability-intensive systems: applications to bug detection and minimization. *Formal Aspects of Computing* 33 (2021), 1147–1172.

[12] Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wasowski. 2015. Family-Based Model Checking Without a Family-Based Model Checker. In *SPIN 2015, Proceedings (LNCS, Vol. 9232)*, Bernd Fischer and Jaco Geldenhuys (Eds.). Springer, 282–299.

[13] Aleksandar S. Dimovski, Sami Lazreg, Maxime Cordy, and Axel Legay. 2023. Family-Based Model Checking of FMultiLTL Properties. In *SPLC 2023, Proceedings* (Tokyo, Japan) *(SPLC '23)*. ACM, New York, NY, USA, 41–51.

[14] Aleksandar S. Dimovski, Axel Legay, and Andrzej Wasowski. 2020. Generalized abstraction-refinement for game-based CTL lifted model checking. *Theor. Comput. Sci.* 837 (2020), 181–206.

[15] Clemens Dubslaff, Christel Baier, and Sascha Klüppelholz. 2015. Probabilistic Model Checking for Feature-Oriented Systems. In *Transactions on Aspect-Oriented Software Development XII*. Springer, Berlin, Germany, 180–220.

[16] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Grady Booch. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Boston, MA, USA.

[17] Paul Gastin and Denis Oddoux. 2001. Fast LTL to Büchi Automata Translation. In *CAV 2001, Proceedings (LNCS, Vol. 2102)*, Gérard Berry, Hubert Comon, and Alain Finkel (Eds.). Springer, 53–65.

[18] Ohad Goudsmid, Orna Grumberg, and Sarai Sheinvald. 2021. Compositional Model Checking for Multi-properties. In *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12597)*, Fritz Henglein, Sharon Shoham, and Yakir Vizel (Eds.). Springer, 55–80. https://doi.org/10.1007/978-3-030-67067-2_4

[19] Holger Hermanns, Björn Wachter, and Lijun Zhang. 2008. Probabilistic CEGAR. In *Computer Aided Verification*, Aarti Gupta and Sharad Malik (Eds.). 162–175.

[20] Gerard J. Holzmann. 2004. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley.

[21] Paulius Juodisius, Atrisha Sarkar, Raghava Rao Mukkamala, Michal Antkiewicz, Krzysztof Czarnecki, and Andrzej Wasowski. 2019. Clafer: Lightweight Modeling of Structure, Behaviour, and Variability. *Art Sci. Eng. Program.* 3, 1 (2019), 2. https://doi.org/10.22152/programming-journal.org/2019/3/2

[22] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21. Carnegie Mellon University.

[23] Jacob Krüger, Sebastian Nielebock, Sebastian Krieter, Christian Diedrich, Thomas Leich, Gunter Saake, Sebastian Zug, and Frank Ortmeier. 2017. Beyond Software Product Lines: Variability Modeling in Cyber-Physical Systems. In *SPLC 2017, Proceedings*, Myra B. Cohen, Mathieu Acher, Lidia Fuentes, Daniel Schall, Jan Bosch, Rafael Capilla, Ebrahim Bagheri, Yingfei Xiong, Javier Troya, Antonio Ruiz Cortés, and David Benavides (Eds.). ACM, 237–241.

[24] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *CAV 2011, Proceedings (LNCS, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 585–591.

[25] Kim Guldstrand Larsen, Axel Legay, and Danny Bøgsted Poulsen. 2023. Refinement of Systems with an Attacker Focus. In *Formal Methods for Industrial Critical Systems - 28th International Conference, FMICS 2023, Antwerp, Belgium, September 20-22, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 14290)*, Alessandro Cimatti and Laura Titolo (Eds.). Springer, 96–112. https://doi.org/10.1007/978-3-031-43681-9_6

[26] Sami Lazreg, Maxime Cordy, and Axel Legay. 2022. Verification of variability-intensive stochastic systems with statistical model checking. In *ISoLA 2022, Proceedings*. Springer, 448–471.

[27] Amir Pnueli. 1977. The Temporal Logic of Programs. In *Annual Symposium on Foundations of Computer Science, 1977*. IEEE Computer Society, 46–57.

[28] Klaus Pohl and Andreas Metzger. 2018. Software Product Lines. In *The Essence of Software Engineering*, Volker Gruhn and Rüdiger Striemer (Eds.). Springer, 185–201. https://doi.org/10.1007/978-3-319-73897-0_11

[29] Fabio Somenzi. 2009. CUDD: CU decision diagram package release 2.4. 2. (2009).

[30] Maurice H. ter Beek, Ferruccio Damiani, Stefania Gnesi, Franco Mazzanti, and Luca Paolini. 2015. From Featured Transition Systems to Modal Transition Systems with Variability Constraints. In *SEFM 2015, Proceedings (LNCS, Vol. 9276)*, Radu Calinescu and Bernhard Rumpe (Eds.). Springer, 344–359.

[31] Maurice H. ter Beek, Erik P. de Vink, and Tim A. C. Willemse. 2017. Family-Based Model Checking with mCRL2. In *FASE 2017, Proceedings (LNCS, Vol. 10202)*, Marieke Huisman and Julia Rubin (Eds.). Springer, 387–405.

[32] Maurice H. ter Beek, Axel Legay, Alberto Lluch-Lafuente, and Andrea Vandin. 2016. Statistical Model Checking for Product Lines. In *ISoLA 2016, Proceedings (LNCS, Vol. 9952)*, Tiziana Margaria and Bernhard Steffen (Eds.). 114–133.

[33] Maurice H. ter Beek, Axel Legay, Alberto Lluch-Lafuente, and Andrea Vandin. 2021. Quantitative Security Risk Modeling and Analysis with RisQFLan. *Comput. Secur.* 109 (2021), 102381.