Family-based model checking of fMultiLTL properties

Anonymous Author(s)
Submission Id: splc23

ABSTRACT

We introduce a new logic for expressing multi-properties of system families (Software Product Lines - SPLs). While the standard LTL logic refers only to a single trace at a time, fMultiLTL logic proposed here refers to multiple traces originating from different sets of variants of the SPL. This is achieved by allowing so-called *featured quantification* over traces, $\forall^{\psi}$ and $\exists^{\psi}$, where the feature expression ψ describes a set of variants (sub-family) the quantified trace comes from. A specialized family-based model checking algorithm for verifying some fragments of fMultiLTL is given. A prototype family-based model checker has been implemented. We illustrate the practicality of this approach on several SPL models.

CCS CONCEPTS

• **Software and its engineering → Software notations and tools**; *Software creation and management*; • **Theory of computation** → Semantics and reasoning.

KEYWORDS

Software Product Lines, Model Checking, LTL, Temporal Multi-Properties,

ACM Reference Format:

Anonymous Author(s). 2023. Family-based model checking of fMultiLTL properties. In *Proceedings of the 27th ACM International Systems and Software Product Lines Conference (SPLC '23), August 28 – September 1, 2023, Tokyo, Japan.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3486609.3487202

1 INTRODUCTION

Software Product Line Engineering (SPLE) [13, 34] represents an efficient method for building families of similar systems. Implementations of such *system families* use *features* (statically configured options) to organize the variable functionality. Family members, called *variants*, are specified in terms of features selected for that particular variant. The reuse of code common to multiple variants is thus maximized. Recently, the SPL method has grown in popularity, especially in the domains of embedded systems, system-level software, communication protocols, etc [13].

In many application domains, such as automotive and avionics, quality assurance is of predominant importance. This requires a solid evidence that system families indeed satisfy their specifications. Researchers have addressed this problem by designing compact representations for modelling the behaviour of all variants of a system family in a single compact structure, and by designing aggregate *family-based model checking* algorithms to efficiently verify such compact representations. In particular, the family-based model checking algorithms allow simultaneous verification of all variants of a system family in a single run by exploiting the commonalities between the variants. Those algorithms are capable of identifying all variants that satisfy a property, as well as all variants that do not satisfy the property together with the corresponding counter-examples. Specialized family-based model checking algorithms have been developed for various modelling formalisms: reactive [9, 10, 16], real-time [14, 23], probabilistic [5] systems, as well as for verification of properties in various temporal logics: LTL [9, 10], CTL* [16], μ-calculus [37], etc.

The linear-time temporal logic (LTL) is a logic for expressing trace properties. However, some behaviors cannot be expressed by referring to each trace individually. For example, secure information flow and non-interference [2, 38] are maintained in a system if for every two traces, if their low-security inputs are identical then so are their low security outputs, regardless of their high-security variables. They cannot be characterized via single traces. In fact, they cannot be expressed neither in CTL* nor in μ-calculus. In [6, 25, 27], properties describing the behaviour of a combination of traces are introduced. They are known as *hyper-properties* (HyperLTL) [6, 25] when different traces refer to the same system, and *multi-properties* (MultiLTL) [27] when different traces refer to different components of a system. That is, MultiLTL enable us to relate traces from one component (sub-system) to traces of another component of a compound system. We now extend the notion of MultiLTL in the context of system families and SPLs, thus obtaining the so-called *featured* MultiLTL, denoted by fMultiLTL.

In this paper, we introduce a new logic fMultiLTL for specifying multi-properties of system families and we study algorithms for their automatic verification. fMultiLTL generalizes LTL by explicitly relating traces from different variants of a system family. While LTL implicitly quantifies over only a single execution trace of a system, fMultiLTL allows explicit quantification over multiple execution traces of a system family simultaneously, as well as propositions that specify relationships among those traces. In particular, fMultiLTL allows featured quantification, $\forall^{\psi}$ and $\exists^{\psi}$, referring to the sub-family (a set of variants) described by the feature expression ψ. This way, traces from the sub-family described by ψ can be referred to in the atomic propositions. Since a system family consists of a set of similar systems, fMultiLTL properties will enable us to relate traces from one subset of systems to another subset. For example, the *diversity* property [35] asks all systems from a family to represent a different implementation of the same high-level system. That is, all systems implement the same functionality but differ in their implementation details. Diversity has been used as a security property to resist attacks that exploit memory layout or instruction sequence specifics. Given a high-level system

SPLC'23, August 28 – September 1, 2023, Tokyo, Japan

ACM ISBN 978-1-4503-9112-2/21/10…$15.00
https://doi.org/10.1145/3486609.3487202

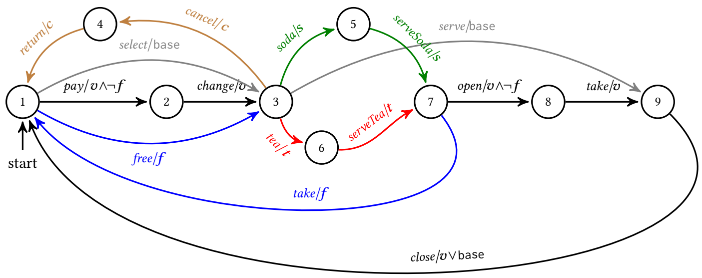

Figure 1: The FTS VendMachine.

described with the base feature, and two low-level implementations described with features f1 and f2 respectively, the diversity property can be expressed as:

$$\varphi_1 = \forall^{\mathtt{base}}_{\pi_0} \exists^{\mathtt{f1}}_{\pi_1} \exists^{\mathtt{f2}}_{\pi_2}.\square(\mathtt{in}_{\pi_0} = \mathtt{in}_{\pi_1} = \mathtt{in}_{\pi_2} \implies \mathtt{out}_{\pi_0} = \mathtt{out}_{\pi_1} = \mathtt{out}_{\pi_2})$$
$$\varphi_2 = \forall^{\mathtt{f1}}_{\pi_1} \exists^{\mathtt{base}}_{\pi_0}.\square(\mathtt{in}_{\pi_1} = \mathtt{in}_{\pi_0} \implies \mathtt{out}_{\pi_1} = \mathtt{out}_{\pi_0})$$
$$\varphi_3 = \forall^{\mathtt{f2}}_{\pi_2} \exists^{\mathtt{base}}_{\pi_0}.\square(\mathtt{in}_{\pi_2} = \mathtt{in}_{\pi_0} \implies \mathtt{out}_{\pi_2} = \mathtt{out}_{\pi_0})$$

where $\mathtt{in}_{\pi_0} = \mathtt{in}_{\pi_1} = \mathtt{in}_{\pi_2}$ and $\mathtt{out}_{\pi_0} = \mathtt{out}_{\pi_1} = \mathtt{out}_{\pi_2}$ express that the three traces π_0, π_1, π_2 agree on the input and output variables in and out, respectively. Note that the traces π_0, π_1, and π_2 come from the systems that contain features base, f1 and f2, respectively. Our fMultiLTL logic enables to directly and naturally express properties like the one above.

We present family-based model checking algorithms applicable to restricted type of fMultiLTL properties, called alternation-free fMultiLTL, in which the series of quantifiers at the beginning of a formula involve zero alternation. Finally, we have implemented within the ProVeLines tool [14] and practically evaluated the algorithms for verifying the alternation-free fragment of fMultiLTL. This is a useful fragment which allows specifying many interesting properties of system families.

To summarize, our contributions are as follows:

- **(1)** We define a new logic fMultiLTL for expressing properties that specify relations over multiple traces from various sets of variants of a system family;
- **(2)** We propose a specialized family-based model checking algorithm for automatic verification of alternation-free fragment of fMultiLTL;
- **(3)** We describe a prototype implementation of our family-based model checking algorithm and use it to verify some interesting alternation-free fMultiLTLproperties of system families.

2 BACKGROUND: SYSTEM FAMILIES

In this section, we summarize the existing background for our work. We present modelling formalisms used to compactly represent system families, and define their semantics.

Let $\mathcal{F} = \{A_1, \ldots, A_n\}$ be a finite set of Boolean variables representing the features available in a system family. A specific subset of features, $k \subseteq \mathcal{F}$, known as *configuration*, specifies a *variant* of a system family. We assume that only a subset $\mathcal{K} \subseteq 2^{\mathcal{F}}$ of configurations are *valid*. An alternative representation of configurations is based upon propositional formulae. Each configuration $k \in \mathcal{K}$ can be represented by a formula: $\nu(A_1) \wedge \ldots \wedge \nu(A_n)$, where $\nu(A_i) = A_i$ if $A_i \in k$, and $\nu(A_i) = \neg A_i$ if $A_i \notin k$ for $1 \leq i \leq n$. We will use both representations interchangeably.

We use *transition systems* (TS) to describe behaviors of single systems. A *transition system* is a tuple $\mathcal{T} = (S, Act, I, trans, AP, L)$, where S is a set of states; $I \subseteq S$ is a set of initial states; $trans \subseteq S \times Act \times S$ is a transition relation which is *total*, so that for each state there is an outgoing transition; AP is a set of atomic propositions; and $L : S \to 2^{AP}$ is a labelling function specifying which atomic propositions hold in a state. We write $s_1 \xrightarrow{\lambda} s_2$ when $(s_1, \lambda, s_2) \in trans$. A *path* of a TS $\mathcal{T}$ is an *infinite* sequence $\rho = s_0 s_1 s_2 \ldots$ with $s_0 \in I$ such that $s_i \xrightarrow{\lambda_{i+1}} s_{i+1}$ for all $i \geq 0$ ($\lambda_{i+1} \in Act$). A *trace* corresponding to the path $\rho = s_0 s_1 s_2 \ldots$ is the sequence of sets of propositions $\mathtt{trace}(\rho) = L(s_0)L(s_1)L(s_2)\ldots$. The *semantics* of the TS $\mathcal{T}$, denoted as $[[\mathcal{T}]]_{TS}$, is the set of its traces.

A *featured transition system* (FTS) represents a compact model, which describes the behavior of a whole family of systems in a single monolithic description. Their transitions are guarded by a *presence condition* that identifies the variants they belong to. The presence conditions ψ are drawn from the set of feature expressions, $FeatExp(\mathcal{F})$, which are propositional logic formulae over $\mathcal{F}$:

$$\psi ::= true \mid A \in \mathcal{F} \mid \neg\psi \mid \psi_1 \wedge \psi_2$$

We write $[[\psi]]$ for the set of configurations that satisfy ψ, i.e. $k \in [[\psi]]$ iff $k \models \psi$.

A *featured transition system* (FTS) is defined to be a tuple $\mathbb{F} = (S, Act, I, trans, AP, L, \mathcal{F}, \mathcal{K}, \gamma)$, where $(S, Act, I, trans, AP, L)$ form a TS; $\mathcal{F}$ is a set of available features; $\mathcal{K}$ is a set of valid configurations; and $\gamma : trans \to FeatExp(\mathcal{F})$ is a total function decorating transitions with presence conditions (feature expressions). The *projection* of an FTS $\mathbb{F}$ to a configuration $k \in \mathcal{K}$, denoted as $Pr_k(\mathbb{F})$, is the TS $(S, Act, I, trans', AP, L)$, where $trans' = \{t \in trans \mid k \models \gamma(t)\}$. We lift the definition of *projection* to sets of configurations $\mathcal{K}' \subseteq \mathcal{K}$, denoted as $Pr_{\mathcal{K}'}(\mathbb{F})$, by keeping the transitions admitted by at least one of the configurations in $\mathcal{K}'$. That is, $Pr_{\mathcal{K}'}(\mathbb{F})$, is the FTS $(S, Act, I, trans', AP, L, \mathcal{F}, \mathcal{K}', \gamma')$, where $trans' = \{t \in trans \mid \exists k \in \mathcal{K}'.k \models \gamma(t)\}$ and $\gamma' = \gamma|_{trans'}$ is the restriction of γ to $trans'$. The *semantics* of an FTS $\mathbb{F}$, denoted as $[[\mathbb{F}]]_{FTS}$, is the union of traces of the projections on all valid variants $k \in \mathcal{K}$, i.e. $[[\mathbb{F}]]_{FTS} = \cup_{k \in \mathcal{K}}[[Pr_k(\mathbb{F})]]_{TS}$. Moreover, the semantics of the projection FTS $Pr_{\mathcal{K}'}(\mathbb{F})$ is $[[Pr_{\mathcal{K}'}(\mathbb{F})]]_{FTS} = \cup_{k \in \mathcal{K}'}[[\pi_k(\mathbb{F})]]_{TS}$.

Example 2.1. The FTS VendMachine in Fig. 1 has features $\mathcal{F} = \{\mathtt{base}, v, t, s, c, f\}$. The feature base is used only for implementing the high-level system and is not present in other configurations. The set of all other valid configurations is obtained by combining the above features (except base). The feature v is for purchasing a drink from the Vending machine; s is for serving Soda; t is for serving Tea; c is for Canceling a purchase after a coin is entered; and f is for offering Free drinks. Each transition is labeled by a feature expression specifying in which variants the transition is included. For instance, the transition ③ $\xrightarrow{soda/s}$ ⑤ is included in variants where feature s is enabled. The feature v is mandatory, and at least one of s or t is enabled in any valid configuration. The set of valid configurations is thus:

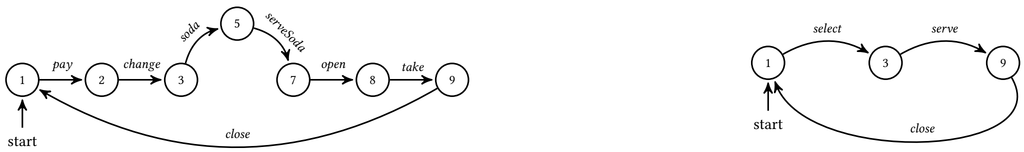

Figure 2: TSs $Pr_{\{v,s\}}$(VendMachine) (left) and $Pr_{\{\text{base}\}}$(VendMachine) (right).

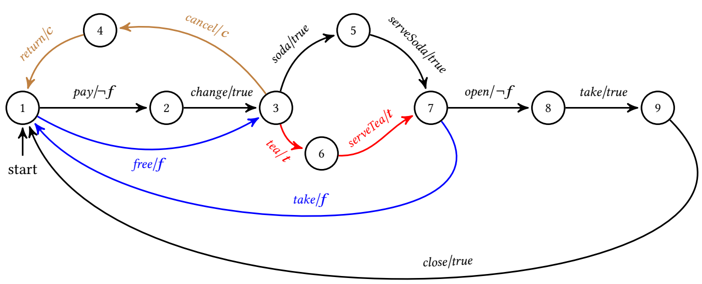

Figure 3: The FTS $Pr_{[[v\wedge s]]}$(VendMachine).

$\mathcal{K}^{\text{VM}} = \{\{\text{base}\}, \{v, s\}, \{v, t\}, \{v, s, t\}, \{v, s, c\}, \{v, t, c\}, \{v, s, t, c\}, \{v, s, f\}, \{v, t, f\}, \{v, s, t, f\}, \{v, s, c, f\}, \{v, t, c, f\}, \{v, s, t, c, f\}\}$.

Figure 2 shows two variants of VendMachine: a version that only serves soda, and a high-level implementation. The former variant is described by the configuration: $\{v, s\}$, equivalently as a formula: $\neg\text{base} \wedge v \wedge s \wedge \neg t \wedge \neg c \wedge \neg f$. The model presented in the figure is obtained by the projection $Pr_{\{v,s\}}$(VendMachine). It accepts payment, returns change, serves a soda, opens the access compartment, so that the user can take the soda, and close it again so that a next user can be served. The latter variant is described by the configuration: $\{\text{base}\}$, equivalently as a formula: $\text{base}\wedge\neg v\wedge\neg s\wedge\neg t\wedge\neg c\wedge\neg f$. Its model is obtained by $Pr_{\{\text{base}\}}$(VendMachine).

On the other hand, note that $[[v \wedge s]] = \{k \in \mathcal{K}^{\text{VM}} \mid k \models v\wedge s\} = \{\{v, s\}, \{v, s, t\}, \{v, s, c\}, \{v, s, t, c\}, \{v, s, f\}, \{v, s, t, f\}, \{v, s, c, f\}, \{v, s, t, c, f\}\}$ represents a sub-family of VendMachine. The FTS $Pr_{[[v\wedge s]]}$(VendMachine) is shown in Fig. 3. Note that transition $① \xrightarrow{select/\text{base}} ③$ is not present in this FTS, since it is not present in any variant from $[[v \wedge s]]$. Also, all literals corresponding to v and s in feature expression are replaced with *true* (see Fig. 3). □

3 FMULTILTL PROPERTIES

We now present *featured* MultiLTL, denoted fMultiLTL, a logic for describing multi-properties of system families described by FTSs. fMultiLTL extends LTL with explicit quantification over traces. It is defined inductively as follows:

$$\varphi ::= \exists^{\psi}\pi.\varphi \mid \forall^{\psi}\pi.\varphi \mid \phi$$
$$\phi ::= a_\pi \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \bigcirc\phi \mid \phi_1 \mathsf{U} \phi_2$$

where π is a trace variable, $\psi \in FeatExp(\mathcal{F})$, and $a \in AP$. Intuitively, $\exists^{\psi}\pi.\varphi$ means that there exists a trace in the sub-family $Pr_{[[\psi]]}(\mathbb{F})$ that satisfies φ, and $\forall^{\psi}\pi.\varphi$ means that φ holds for every trace in $Pr_{[[\psi]]}(\mathbb{F})$. Atomic propositions $a \in AP$ are annotated with trace variables π, denoted a_π, to disambiguate to which trace the proposition refers to. A formula φ is *closed* if all trace variables π are in the scope of a quantifier. Boolean connectives disjunction ($\vee$), implication ($\implies$), and equivalence ($\equiv$) are defined as syntactic sugar. The other temporal operators are also defined by means of syntactic sugar, for instance: $\Diamond\phi = true\,\mathsf{U}\phi$ (ϕ holds eventually) and $\Box\phi = \neg\Diamond\neg\phi$ (ϕ always holds).

Formally, the semantics of fMultiLTL is defined as follows. Let $Tr \subseteq (2^{AP})^\omega$ be a set of all traces and let $t \in Tr$ be a trace. We use $t[i]$ to denote the i-th element of t. We write $t[0, i]$ to denote the prefix of t up to and including i-th element, and $t[i, \infty]$ to denote the infinite suffix of t beginning with i-th element. Let V be a set of trace variables, and $\Pi : V \to Tr$ be a trace assignment. Let $\Pi[\pi \mapsto t]$ be the function obtained from Π, by mapping π to t. Let Π^i be the function defined by $\Pi^i(\pi) = (\Pi(\pi))[i, \infty]$. Satisfaction of a formula φ for an FTS $\mathbb{F}$ and a trace assignment Π is defined as:

$$\Pi \models_{\mathbb{F}} \exists^{\psi}\pi.\varphi \text{ iff } \exists t \in [[Pr_{[[\psi]]}(\mathbb{F})]]_{\text{FTS}}.\Pi[\pi \mapsto t] \models_{\mathbb{F}} \varphi$$
$$\Pi \models_{\mathbb{F}} \forall^{\psi}\pi.\varphi \text{ iff } \forall t \in [[Pr_{[[\psi]]}(\mathbb{F})]]_{\text{FTS}}.\Pi[\pi \mapsto t] \models_{\mathbb{F}} \varphi$$
$$\Pi \models_{\mathbb{F}} a_\pi \text{ iff } a \in \Pi(\pi)[0]$$
$$\Pi \models_{\mathbb{F}} \neg\phi \text{ iff } \Pi \not\models_{\mathbb{F}} \phi$$
$$\Pi \models_{\mathbb{F}} \phi_1 \wedge \phi_2 \text{ iff } \Pi \models_{\mathbb{F}} \phi_1 \text{ and } \Pi \models_{\mathbb{F}} \phi_2$$
$$\Pi \models_{\mathbb{F}} \bigcirc\phi \text{ iff } \Pi^1 \models_{\mathbb{F}} \phi$$
$$\Pi \models_{\mathbb{F}} (\phi_1 \mathsf{U} \phi_2) \text{ iff } \exists i \geq 0.\, \left(\Pi^i \models_{\mathbb{F}} \phi_2 \wedge \forall j. 0 \leq j < i. \Pi^j \models_{\mathbb{F}} \phi_1\right)$$

A FTS $\mathbb{F}$ satisfies a closed formula φ, written $\mathbb{F} \models \varphi$, if $\Pi_\emptyset \models_{\mathbb{F}} \varphi$ where $\Pi_\emptyset$ is the trace assignment with empty domain.

Example 3.1. Let us consider the VendMachine of Fig. 1. Assume that the atomic proposition start holds in state ①, whereas served holds in state ⑨. Consider the following properties:

$$\varphi_1 = \forall^{\text{base}}_{\pi_0}\exists^{v\wedge s}_{\pi_1}\exists^{v\wedge t}_{\pi_2}.\Box(\text{start}_{\pi_0} \wedge \text{start}_{\pi_1} \wedge \text{start}_{\pi_2} \implies \Diamond\text{served}_{\pi_0} \wedge \Diamond\text{served}_{\pi_1} \wedge \Diamond\text{served}_{\pi_2})$$
$$\varphi_2 = \forall^{\text{base}}_{\pi_0}\forall^{v\wedge s}_{\pi_1}\forall^{v\wedge t}_{\pi_2}.\Box(\text{start}_{\pi_0} \wedge \text{start}_{\pi_1} \wedge \text{start}_{\pi_2} \implies \Diamond\text{served}_{\pi_0} \wedge \Diamond\text{served}_{\pi_1} \wedge \Diamond\text{served}_{\pi_2})$$

The formula φ_1 states that for every trace π_0 form the base variant, there are traces π_1 and π_2 from $[[v \wedge s]]$ and $[[v \wedge t]]$ sub-families, such that after the corresponding machines have been *started*, they will eventually *serve* the drink to the customer in all three traces. The formula φ_2 requires the above property to hold for all triples of traces from base, $[[v \wedge s]]$ and $[[v \wedge t]]$ sub-families.

The formula φ_1 holds in the VendMachine, but the formula φ_2 is violated. This is due to the fact that there are traces $t_1 = ① \to ② \to ③ \to ①$ and $t_2 = ① \to ③ \to ⑤ \to ⑦ \to ①$, which belong to both $[[Pr_{[[v\wedge s]]}(\text{VendMachine})]]_{FTS}$ as well as $[[Pr_{[[v\wedge t]]}(\text{VendMachine})]]_{FTS}$, such that they do not visit the state ⑨ where served holds. In particular, we have that $t_1 \in [[Pr_{\{v,s,c\}}(\text{VendMachine})]]_{TS}$, $t_2 \in [[Pr_{\{v,t,f\}}(\text{VendMachine})]]_{TS}$. □

4 FAMILY-BASED MODEL CHECKING ALGORITHM

We present a family-based model checking algorithm for the alternation-free fragment of fMultiLTL, called fMultiLTL$_1$, in which the series of quantifiers at the beginning of a formula involve zero alternation. We assume the fMultiLTL$_1$ formula to be of the form $\forall^{\psi_1}\pi_1 \ldots \forall^{\psi_n}\pi_n.\phi$. Formulas of the form $\exists^{\psi_1}\pi_1 \ldots \exists^{\psi_n}\pi_n.\phi$ can be rewritten as $\forall^{\psi_1}\pi_1 \ldots \forall^{\psi_n}\pi_n.\neg\phi$. Our algorithm extends the standard automata-theoretic approach to model checking [1, 39]. Hence, it uses various automata constructions [39], language non-emptiness, self-composition [2, 38], and a projection operator.

Büchi automata. Büchi automata (BA) [1, 39] are finite-state automata that accept words of infinite length. A BA is a tuple $A = (Q, \Sigma, \delta, Q_0, F)$ where Q is a set of states, Σ is an alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation, $Q_0 \subseteq Q$ is a set of initial states, and $F \subseteq Q$ is a set of accepting states. A path $q_0q_1 \ldots \in Q^\omega$ of a BA is over a word $w = \alpha_1\alpha_2 \ldots \in \Sigma^\omega$, if for all $i \geq 0$, $(q_i, \alpha_{i+1}, q_{i+1}) \in \delta$. A word w is *recognized* by a BA A if there exists a path over the word w with some accepting states from F occurring infinitely often. The *language* $\mathcal{L}(A)$ of a BA A is the set of words that the automaton A recognizes.

Composition. The n-fold composition of FTSs $\mathbb{F}_1, \ldots, \mathbb{F}_n$ is the synchronous product $\mathbb{F}_1 \otimes \ldots \otimes \mathbb{F}_n$. Given n FTSs defined as $\mathbb{F}_i = (S_i, Act_i, I_i, trans_i, AP, L_i, \mathcal{F}, \mathcal{K}_i, \gamma_i)$ for $1 \leq i \leq n$, we define the composition $\mathbb{F}_1 \otimes \ldots \otimes \mathbb{F}_n$ as the FTS $(S_1 \times \ldots \times S_n, Act_1 \times \ldots \times Act_n, I_1 \times \ldots \times I_n, trans, AP^n, L, \mathcal{F}, \mathcal{K}_1 \times \ldots \times \mathcal{K}_n, \gamma_1 \times \ldots \times \gamma_n)$ such that for all states $(s_1, \ldots, s_n), (t_1, \ldots, t_n)$ and actions $(\lambda_1, \ldots, \lambda_n)$, we have $(s_1, \ldots, s_n) \xrightarrow{(\lambda_1,\ldots,\lambda_n)} (t_1, \ldots, t_n) \in trans$ iff $s_i \xrightarrow{\lambda_i} t_i \in trans_i$ for all $1 \leq i \leq n$. Moreover, $L : S_1 \times \ldots \times S_n \rightarrow 2^{AP^n}$ such that $Proj_i(L(s_1, \ldots, s_n)) \subseteq L_i(s_i)$ for all $1 \leq i \leq n$, where $Proj_i$ is the set obtained by projecting a set of n-tuples to their i-th components. Finally, $\gamma_1 \times \ldots \times \gamma_n\big((s_1, \ldots, s_n) \xrightarrow{(\lambda_1,\ldots,\lambda_n)} (t_1, \ldots, t_n)\big) = (\psi_1, \ldots, \psi_n)$ if $\gamma_i(s_i \xrightarrow{\lambda_i} t_i) = \psi_i$ for $1 \leq i \leq n$. The projection of $\mathbb{F}_1 \otimes \ldots \otimes \mathbb{F}_n$ to a configuration $(k_1, \ldots, k_n) \in \mathcal{K}_1 \times \ldots \times \mathcal{K}_n$, denoted as $Pr_{(k_1,\ldots,k_n)}(\mathbb{F}_1 \otimes \ldots \otimes \mathbb{F}_n)$ is the TS obtained by restricting the transitions of $\mathbb{F}_1 \otimes \ldots \otimes \mathbb{F}_n$ to only those whose feature expressions $(\psi_1, \ldots, \psi_n)$ are satisfied by $(k_1, \ldots, k_n)$. The semantics $[[\mathbb{F}_1 \otimes \ldots \otimes \mathbb{F}_n]]_{FTS}$ is $\cup_{(k_1,\ldots,k_n) \in \mathcal{K}_1 \times \ldots \times \mathcal{K}_n} [[Pr_{(k_1,\ldots,k_n)}(\mathbb{F}_1 \otimes \ldots \otimes \mathbb{F}_n)]]_{TS}$. Let `zip` denote the function that maps an n-tuple of sequences to a single sequence of n-tuples. For example, $\texttt{zip}([1, 3, 5, \ldots], [2, 4, 6, \ldots]) = [(1, 2), (3, 4), (5, 6), \ldots]$. Let `unzip` denote its inverse function. Hence, $\mathbb{F}_1 \otimes \ldots \otimes \mathbb{F}_n$ contains a trace $\texttt{zip}(t_1, \ldots, t_n)$ if $\mathbb{F}_1, \ldots, \mathbb{F}_n$ contain traces $t_1, \ldots, t_n$, respectively. That is,

$$[[\mathbb{F}_1 \otimes \ldots \otimes \mathbb{F}_n]]_{FTS} = \{\texttt{zip}(t_1, \ldots, t_n) \mid t_i \in [[\mathbb{F}_i]]_{FTS} \text{ for } 1 \leq i \leq n\}$$

Given an FTS $\mathbb{F}$, we write $\mathbb{F}^{\psi_1 \otimes \ldots \otimes \psi_n}$ for the composition $Pr_{[[\psi_1]]}(\mathbb{F}) \otimes \ldots \otimes Pr_{[[\psi_n]]}(\mathbb{F})$.

Formula-to-automaton construction. Suppose a fMultiLTL$_1$ formula $\forall^{\psi_1}\pi_1 \ldots \forall^{\psi_n}\pi_n.\phi$ is given. We construct a generalized BA $A_\phi = (Q_\phi, \Sigma_\phi, \delta_\phi, Q^0_\phi, F_\phi)$ for ϕ. A generalized BA is the same as a BA except that it has a multiple of accepting states [1]. First, we preprocess ϕ to put it in a negation normal form (NNF) [1]. To construct the states of A_ϕ, we define $\texttt{closure}(\phi)$ to be the set of all sub-formulae of ϕ and their negations. Then we define elementary sets of formulae $B \subseteq \texttt{closure}(\phi)$ that are maximal consistent sets with respect to ϕ [1]. When we construct elementary sets of formulae of $\texttt{closure}(\phi)$, we generate n-tuples of all atomic propositions that are in that elementary set corresponding to traces $\pi_1, \ldots, \pi_n$. The set of states Q_ϕ is the set of elementary sets of formulae of $\texttt{closure}(\phi)$ [1]. Intuitively, a state describes a set of trace tuples where each tuple satisfies all formulae in the elementary set representing that state. The initial set of states is $Q^0_\phi = \{B \in Q_\phi \mid \phi \in B\}$. The alphabet is $\Sigma_\phi = (2^{AP})^n$, so each letter of the alphabet is an n-tuple of sets of atomic propositions. The transition relation $\delta_\phi : Q_\phi \times \Sigma_\phi \times Q_\phi$ is given by: if $A = B \cap (AP \cup \{\emptyset\})^n$, then $\delta_\phi(B, A)$ is a straightforward extension to n-tuples of the standard definition of δ_ϕ for LTL [1]. If $A \neq B \cap (AP \cup \{\emptyset\})^n$, then $\delta_\phi(B, A) = \emptyset$. The set of accepting states F_ϕ contains one set $\{B \in Q_\phi \mid \neg(\phi_1 \mathsf{U} \phi_2) \in B \text{ or } \phi_2 \in B\}$ for each until formula $(\phi_1 \mathsf{U} \phi_2)$ in $\texttt{closure}(\phi)$.

The BA A_ϕ accepts exactly the words $w \in \mathcal{L}(A_\phi)$, which are sequences of n-tuples, for which $\Pi \models_\emptyset \phi$, where $\Pi = [\pi_1 \mapsto proj_1(\texttt{unzip}(w))] \ldots [\pi_n \mapsto proj_n(\texttt{unzip}(w))]$ (where $proj_i$ denotes the projection of an n-tuple to its i-th component) and $\emptyset$ is the empty FTS. The construction closely follows the standard LTL automata construction [39], with addition that now we work with n-tuple words. In particular, Σ_ϕ is $(2^{AP})^n$, so each letter is a n-tuple of sets of atomic propositions.

Example 4.1. Consider the formula $\forall^{f_1}\pi_1\forall^{f_2}\pi_2. \bigcirc (a_{\pi_1} \wedge a_{\pi_2})$, where $\phi = \bigcirc(a_{\pi_1} \wedge a_{\pi_2})$. We have

$$\texttt{closure}(\phi) = \{a_{\pi_1}, a_{\pi_2}, \neg a_{\pi_1}, \neg a_{\pi_2}, a_{\pi_1} \wedge a_{\pi_2}, \neg(a_{\pi_1} \wedge a_{\pi_2}), \phi, \neg\phi\}$$

The state space Q_ϕ consists of the following elementary sets:

$$B_1 = \{(a_{\pi_1}, a_{\pi_2}), a_{\pi_1} \wedge a_{\pi_2}, \phi\} \qquad B_2 = \{(a_{\pi_1}, a_{\pi_2}), a_{\pi_1} \wedge a_{\pi_2}, \neg\phi\}$$
$$B_3 = \{(a_{\pi_1}, \emptyset), \neg(a_{\pi_1} \wedge a_{\pi_2}), \phi\} \qquad B_4 = \{(a_{\pi_1}, \emptyset), \neg(a_{\pi_1} \wedge a_{\pi_2}), \neg\phi\}$$
$$B_5 = \{(\emptyset, a_{\pi_2}), \neg(a_{\pi_1} \wedge a_{\pi_2}), \phi\} \qquad B_6 = \{(\emptyset, a_{\pi_1}), \neg(a_{\pi_1} \wedge a_{\pi_2}), \neg\phi\}$$
$$B_7 = \{(\emptyset, \emptyset), \neg(a_{\pi_1} \wedge a_{\pi_2}), \phi\} \qquad B_8 = \{(\emptyset, \emptyset), \neg(a_{\pi_1} \wedge a_{\pi_2}), \neg\phi\}$$

The initial states are the states that contain ϕ, $Q^0_\phi = \{B_1, B_3, B_5, B_7\}$. $\delta_\phi(B_1, \{(a_{\pi_1}, a_{\pi_2})\}) = \delta_\phi(B_3, \{(a_{\pi_1}, \emptyset)\}) = \delta_\phi(B_5, \{(\emptyset, a_{\pi_2})\}) = \delta_\phi(B_7, \{(\emptyset, \emptyset)\}) = \{B_1, B_2\}$, and we have $\delta_\phi(B_2, \{(a_{\pi_1}, a_{\pi_2})\}) = \delta_\phi(B_4, \{(a_{\pi_1}, \emptyset)\}) = \delta_\phi(B_6, \{(\emptyset, a_{\pi_2})\}) = \delta_\phi(B_8, \{(\emptyset, \emptyset)\}) = \{B_3, B_4, B_5, B_6, B_7, B_8\}$. Note that $\phi = \bigcirc(a_{\pi_1} \wedge a_{\pi_2}) \in B_1, B_3, B_5, B_7$, so in their next states $(a_{\pi_1} \wedge a_{\pi_2})$ should hold, and B_1 and B_2 are the only states that contain $(a_{\pi_1} \wedge a_{\pi_2})$. Similarly, $\neg\phi = \neg \bigcirc (a_{\pi_1} \wedge a_{\pi_2}) \in B_2, B_4, B_6, B_8$, so in their next states $\neg(a_{\pi_1} \wedge a_{\pi_2})$ should hold, and B_3, B_4, B_5, B_6, B_7 and B_8 are the states that contain $\neg(a_{\pi_1} \wedge a_{\pi_2})$. There are no outgoing transitions on other letters. The set F_ϕ is empty as ϕ does not contain an until operator, so every infinite run is accepting. □

Synchronous product. For an FTS $\mathbb{F} = (S, Act, I, trans, AP, L, \mathcal{F}, \mathcal{K}, \gamma)$ and a BA $A = (Q, 2^{AP}, \delta, Q_0, F)$, the *synchronous product* is an FTS $\mathbb{F} \otimes A = (S \times Q, Act, trans', I', AP', L', \mathcal{F}, \mathcal{K}, \gamma')$, where $AP' = Q$ and $L'(s, q) = q$, $(s, q) \xrightarrow{\alpha}{}' (t, p)$ iff $s \xrightarrow{\alpha} t$ and $q \xrightarrow{L(t)} p$, $\gamma'((s, q) \xrightarrow{\alpha}{}' (t, p)) = \gamma(s \xrightarrow{\alpha} t)$, $I' = \{(s_0, q) \mid s_0 \in I, \exists q_0 \in Q_0.(q_0, L(s_0), q) \in \delta\}$.

The algorithm checks $\mathbb{F} \models \forall^{\psi_1}\pi_1 \ldots \forall^{\psi_n}\pi_n.\phi$.

1. We construct the FTS $\mathbb{F}^{\psi_1 \otimes \ldots \otimes \psi_n}$.
2. We construct the Büchi automata A_ϕ and $A_{\neg\phi}$.
3. We construct the FTS $\mathbb{F}^{\psi_1 \otimes \ldots \otimes \psi_n} \otimes A_{\neg\phi}$ and a featured Büchi automata $BA(\mathbb{F}^{\psi_1 \otimes \ldots \otimes \psi_n}) \cap A_\phi$.
4. We check the persistence property $\mathbb{F}^{\psi_1 \otimes \ldots \otimes \psi_n} \otimes A_{\neg\phi} \models \Diamond\Box\neg F$, where F is the set of accepting states of $A_{\neg\phi}$. If the persistence property does not hold, then it corresponds to a counterexample showing that the given fMultiLTL formula is violated by $\mathbb{F}$. Otherwise, if the persistence property holds, we conclude that the given fMultiLTL formula holds.

Figure 4: The family-based model checking algorithm.

Model checking results. Our results for family-based model checking of fMultiLTL$_1$ adapt the corresponding results for verification of HyperLTL$_1$ given in [6].

Theorem 4.2 (fMultiLTL$_1$). *$\mathbb{F} \models \forall^{\psi_1}\pi_1 \ldots \forall^{\psi_n}\pi_n.\phi$ iff $[[\mathbb{F}^{\psi_1 \otimes \ldots \otimes \psi_n} \otimes A_{\neg\phi}]]_{FTS} = \emptyset$*

Proof.

$\forall^{\psi_1}\pi_1 \ldots \forall^{\psi_n}\pi_n.\phi$ does not hold on $\mathbb{F}$
iff
there exists a n-tuple $\Pi_n \in [[\mathbb{F}^{\psi_1 \otimes \ldots \otimes \psi_n}]]_{FTS}$ s.t. $\Pi_n \models_\emptyset \neg\phi$
iff
$[[\mathbb{F}^{\psi_1 \otimes \ldots \otimes \psi_n} \otimes A_{\neg\phi}]]_{FTS}$ is not empty.

□

Algorithm. Our algorithm for verifying fMultiLTL$_1$ adapts the classical automata-theoretic LTL model checking algorithm [1, 39]. To determine whether an FTS $\mathbb{F}$ satisfies a formula $\forall^{\psi_1}\pi_1 \ldots \forall^{\psi_n}\pi_n.\phi$, we call the family-based model checking algorithm illustrated in Fig. 4.

The algorithm in Fig. 4 uses the result from Theorems 4.2 to check fMultiLTL$_1$ formulae. It checks the persistence property $\mathbb{F} \otimes BA \models \Diamond\Box\neg F$, where F is the set of final (accepting) states in the Büchi automaton BA. This reduces to checking if there is a reachable accepting state on a cycle in the FTS $\mathbb{F} \otimes BA$. This is implemented with a double DFS (depth-first search): the outer DFS finds a reachable accepting state, the inner DFS checks whether it is reachable from itself. Both DFS compute the reachability relation of an FTS, and their detailed implementation, denoted `CheckPersistence`, is given in [11, 12]. The procedure `CheckPersistence` is based on computing the *reachability relation* of an FTS $\mathbb{F}$, denoted by $R : S \rightarrow \mathcal{P}(\mathcal{K})$, such that for all states $s \in S$, $k \in R(s)$ iff state s is reachable in the variant $Pr_k(\mathbb{F})$ for configuration k. This procedure generalises the standard DFS algorithm for transition systems, by marking states with sets of configurations, rather than Boolean *visited* flags. In contrast to the standard DFS for transition systems, where no state is visited twice, this feature-aware DFS can visit states multiple times. When $R(s) = \mathcal{K}'$ and the DFS arrives at state s for the second time with a set of configurations $\mathcal{K}''$, such that $\mathcal{K}'' \nsubseteq \mathcal{K}'$, then s although already visited, has to be re-explored. This is because some transitions that were disallowed for $\mathcal{K}'$ in s might be allowed for $\mathcal{K}''$. See [11, 12] for more details.

5 IMPLEMENTATION

We have implemented a prototype tool for verifying fMultiLTL$_1$ formulae as an extension of the ProVeLines tool [14]. ProVeLines is the SPL model checking toolset based on FTSs, which integrates various formalisms for verifying family systems (e.g., reactive and real-time systems; boolean and numerical features; CNF and BDD representation of feature expressions; etc). It uses two modelling languages for SPL specification: *f*Promela [9] is a high-level modelling language for describing system families, and TVL [8] is a textual language for describing sets of features $\mathcal{F}$ and valid configurations $\mathcal{K}$.

*f*Promela is obtained from Promela [28] by adding *feature variables* $\mathcal{F}$ and *guarded-by-features statements* "gd". Promela is a non-deterministic modelling language of the SPIN model checker [28] designed for describing systems composed of concurrent processes that communicate asynchronously. The feature variables, $\mathcal{F}$, used in an *f*Promela model are declared as fields of the special type `features`. The new guarded-by-features statement introduced in *f*Promela is of the form:

$$\text{gd} :: \psi_1 \Rightarrow stm_1 \ldots :: \psi_n \Rightarrow stm_n :: \text{else} \Rightarrow stm \, \text{dg}$$

where $\psi_1, \ldots, \psi_n$ are feature expressions defined over $\mathcal{F}$. The "gd" is a non-deterministic statement similar to "if", except that only feature variables can be used in conditions (guards). It nondeterministically executes the statement stm_i for which the guard ψ_i evaluate to *true* for the current evaluation of feature variables. If none of guards $\psi_1, \ldots, \psi_n$ is *true*, then the `else` statement stm is chosen. Hence, "gd" in *f*Promela plays the same role as "`#ifdef`" in C/CPP SPLs [31].

Fig. 5 shows simple *f*Promela and TVL models. After declaring feature variables `B1` and `B2` as well as the global variables `n` and `i` in the *f*Promela model in Fig. 5 (left), the process `foo` is defined. The statement '`do :: break :: n++ od`' is used to non-deterministically initialize variables `n` and `i` of type `byte` to any integer value from their domain $[0, 255]$ at label `Start`. The first `gd` statement specifies that `i=i+2` is available for variants that contain the feature `B1`, and `skip` for variants with ¬`B1`. The second `gd` statement is similar, except that the guard is the feature `B2`. It states that `i=i+1` is available for variants containing `B1` and `skip` for variants with ¬`B1`. Finally, we print out the current value of `i` at label `Final`. The TVL model in Fig. 5 (right) specifies four valid configurations: {`Main`}, {`Main`, `B1`}, {`Main`, `B2`}, {`Main`, `B1`, `B2`} for this system family. Finally, we specify fMultiLTL properties:

$$\varphi_1 = \forall^{\mathtt{B1}}_{\pi_1}\forall^{\mathtt{B2}}_{\pi_2}.(\mathtt{Start} \wedge \mathtt{n}_{\pi_1} = \mathtt{n}_{\pi_2}) \implies \Diamond(\mathtt{Final} \wedge \mathtt{i}_{\pi_1} \geq \mathtt{i}_{\pi_2})$$
$$\varphi_2 = \exists^{\mathtt{B1}}_{\pi_1}\exists^{\mathtt{B2}}_{\pi_2}.(\mathtt{Start} \wedge \mathtt{n}_{\pi_1} = \mathtt{n}_{\pi_2}) \implies \Diamond(\mathtt{Final} \wedge \mathtt{i}_{\pi_1} \geq \mathtt{i}_{\pi_2})$$

The property φ_1 states that for all traces π_1 from the sub-family $[[\mathtt{B1}]]$ and π_2 from $[[\mathtt{B2}]]$, if the value of `n` in the label `Start` is the same in traces π_1 and π_2, then eventually $\mathtt{i}_{\pi_1} \geq \mathtt{i}_{\pi_2}$ will hold in label `Final`. The property φ_1 does not hold. The counter-example for φ_1 contains a trace $\pi_1 \in Pr_{\mathtt{B1} \wedge \neg \mathtt{B2}}(\mathbb{F}) \subseteq Pr_{[[\mathtt{B1}]]}(\mathbb{F})$ (where $\mathbb{F}$ is the FTS for *f*Promela model in Fig. 5), in which `i`=n+2 in label `Final` (where n is the initial value of variable `n`), and a trace $\pi_2 \in Pr_{\mathtt{B1} \wedge \mathtt{B2}}(\mathbb{F}) \subseteq Pr_{[[\mathtt{B2}]]}(\mathbb{F})$, in which `i`=$n$+3 in label `Final`.

```
0 typedef features {
1   bool B1; bool B2; }
2 features f;
3 byte n, i;
4 active proctype foo() {
5   do :: break :: n++ od;
6   Start: i := n;
7   gd :: f.B1 ⇒ i=i+2 :: else ⇒ skip dg;
8   gd :: f.B2 ⇒ i=i+1 :: else ⇒ skip dg;
9   Final: printf("i : %d", i);
10 }

11 A{p1}[B1] A{p2}[B2]((foo@Start && n{p1}==n{p2}) →
11          ◇(foo@Final && i{p1} ≥ i{p2}))
```

```
0 root Main {
1   group allOf {
2     opt B1,
3     opt B2
4   } }
```

Figure 5: Simple *f*Promela (left) and TVL (right) models

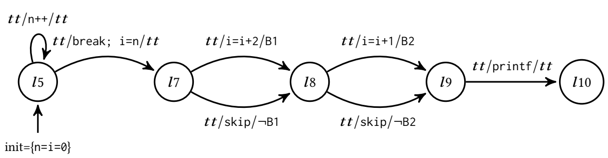

Figure 6: An FPG. The state "*lx*" refers to the line number *x* in the model *foo* in Fig. 5, and *tt* is short for *true*.

The property φ_2 states that there exist traces π_1 from [[B1]] and π_2 from [[B2]], if the value of n in Start is the same in π_1 and π_2, then eventually $\mathtt{i}_{\pi_1} \geq \mathtt{i}_{\pi_2}$ in Final. The property φ_2 holds, and the witness is a trace $\pi_1 \in Pr_{\mathsf{B1}\wedge\mathsf{B2}}(\mathbb{F}) \subseteq Pr_{[[\mathsf{B1}]]}(\mathbb{F})$ in which i=n+3 in Final and a trace $\pi_2 \in Pr_{\neg\mathsf{B1}\wedge\mathsf{B2}}(\mathbb{F}) \subseteq Pr_{[[\mathsf{B2}]]}(\mathbb{F})$ in which i=n+1 in Final. We verify ϕ_2 by encoding it as $\phi_2' = \forall^{\mathsf{B1}}_{\pi_1}\forall^{\mathsf{B2}}_{\pi_2}.\neg\big((\mathtt{Start} \wedge \mathtt{n}_{\pi_1} = \mathtt{n}_{\pi_2}) \implies \Diamond(\mathtt{Final} \wedge \mathtt{i}_{\pi_1} \geq \mathtt{i}_{\pi_2})\big)$, which is equivalent to $\forall^{\mathsf{B1}}_{\pi_1}\forall^{\mathsf{B2}}_{\pi_2}.(\mathtt{Start} \wedge \mathtt{n}_{\pi_1} = \mathtt{n}_{\pi_2}) \wedge \Box(\neg\mathtt{Final} \vee \mathtt{i}_{\pi_1} < \mathtt{i}_{\pi_2})$. A negative answer to ϕ_2' represents a positive answer to ϕ_2, and vice versa. That is, the counter-example violating ϕ_2' represents a witness showing that ϕ_2 is correct.

We now give a brief overview of the *f*Promela semantics [9]. Similarly as a Promela model defines a program graph (PG) [1], an *f*Promela model defines a so-called featured program graph (FPG) [9] that formalizes the control flow of the model. The vertices of the graph are control locations and transitions are annotated with *condition/effect/feature expression* triples. The "gd" statement specifies the feature expression part of transitions. The *semantics* of an FPG is an *FTS* obtained from "*unfolding*" the graph (see [1, Sect. 2] for details). The FPG of our *f*Promela model in Fig. 5 is shown in Fig. 6. The unfolded FTS can be easily constructed, such that each state in it contains the information about the control location (line number) and the current value of variables n and i.

The family-based model checking algorithm is executed *on-the-fly*, by constructing the product FTS $\mathbb{F} \otimes BA$ "on-demand", where $\mathbb{F}$ is the FTS of the system family and BA is the Büchi automaton of the negated formula we consider. The generation of reachable states of $\mathbb{F}$ proceeds in parallel with the construction of the relevant fragment of BA. When generating the successors of a state in BA, we only need to consider the successors matching the current state of $\mathbb{F}$. Hence, we can find an accepting state of BA on a cycle, without the need to generate the entire BA.

6 EVALUATION

We now evaluate our approach for family-based model checking of fMultiLTL$_1$ properties using the ProVeLines tool [14]. The evaluation aims to show that we can verify some interesting properties over model families that are not expressible in the existing logics. Moreover, we want to test and determine the performance limits of the current implementation, and so set the scene for improvements and extensions of our approach in future.

6.1 Experimental setup

Experiments are executed on a 64-bit Intel®Core^{TM} i7-1165G7 CPU@2.80GHz, VM LUbuntu 20.10, with 8 GB memory, and we use a timeout value of 300 seconds. All times are reported as average over five independent executions. The implementation, benchmarks, and all obtained results is available from: link-to-repository-removed-for-double-blind-review. For each experiment, we report: Time which is the time to model check in seconds (this includes the times to parse the *f*Promela model, to build the initial FTS, and to run the model checking algorithm); and Space which is the memory occupied in MB to perform the given model checking task. The evaluation is performed on two benchmarks: WarmingUp and MinePump family-models [9, 10].

6.2 Warming-up example

Combinatorically, the number of variants in $\mathcal{K}$ grows exponentially with the number of features $|\mathcal{F}|$, which means that there is an exponential blow-up in the model checking strategy for LTL that verifies all variants one by one. Although, ProVeLines implements specialized family-based model checking algorithms of LTL that check all variants simultaneously in a single run, its performance still depends on the size and complexity of the configuration space $\mathcal{K}$. Unfortunately, model checking of fMultiLTL is even harder than LTL because, another source of complexity is stemming from the n-fold composition operator and the need to work with n-sized tuples. The size of the synchronous product (n-fold composition)

$\|\mathcal{F}\|$	$\|Q\| = 2$		$\|Q\| = 4$		$\|Q\| = 6$		$\|Q\| = \|\mathcal{F}\|$	
	Time	Space	Time	Space	Time	Space	Time	Space
6	0.097	15.0	0.112	17.7	0.163	28.9	0.163	28.9
7	0.098	15.1	0.114	18.5	0.188	33.2	0.435	69.6
8	0.097	15.2	0.127	19.6	0.219	37.9	0.869	139.8
9	0.100	15.3	0.125	20.2	0.250	42.8	1.708	308.6
10	0.103	15.5	0.129	21.2	0.261	48.2	3.637	716.4
11	0.104	15.7	0.142	21.5	0.292	53.8	8.755	1699.2

Figure 7: Verification of the property φ_1 of the WarmingUp example. Time in sec and Space in MB.

increases exponentially with the number of copies. Thus, reasoning on the product model becomes computationally very prohibitive.

As an experiment, we have tested the limits of our family-based model checking algorithm for fMultiLTL$_1$. We have gradually added variability to the model family in Fig. 5, and we have also generated bigger fMultiLTL$_1$ formulae with bigger number of quantifiers. We write $|Q|$ to denote the number of quantifiers in a fMultiLTL$_1$ formula. This was done by adding unconstrained optional features and by sequentially composing gd statements guarded by all existing features. Note that we have $\mathcal{K} = 2^{|\mathcal{F}|}$, since all features are optional. For example, the fPromela process foo with three features B1, B2, and B3 is:

```
do :: break :: n++ od;
Start: i := n;
gd :: f.B1 ⇒ i=i+3 :: else ⇒ skip dg;
gd :: f.B2 ⇒ i=i+2 :: else ⇒ skip dg;
gd :: f.B3 ⇒ i=i+1 :: else ⇒ skip dg;
Final: printf("i : %d", i);
```

and the corresponding properties with three quantifiers are:

$$\varphi_1 = \forall^{B1}_{\pi_1}\forall^{B2}_{\pi_2}\forall^{B3}_{\pi_3}.(\mathtt{Start} \land \mathtt{n}_{\pi_1} = \mathtt{n}_{\pi_2} = \mathtt{n}_{\pi_3}) \implies \Diamond(\mathtt{Final} \land \mathtt{i}_{\pi_1} \geq \mathtt{i}_{\pi_2} \land \mathtt{i}_{\pi_2} \geq \mathtt{i}_{\pi_3})$$
$$\varphi_2 = \exists^{B1}_{\pi_1}\exists^{B2}_{\pi_2}\exists^{B3}_{\pi_3}.(\mathtt{Start} \land \mathtt{n}_{\pi_1} = \mathtt{n}_{\pi_2} = \mathtt{n}_{\pi_3}) \implies \Diamond(\mathtt{Final} \land \mathtt{i}_{\pi_1} \geq \mathtt{i}_{\pi_2} \land \mathtt{i}_{\pi_2} \geq \mathtt{i}_{\pi_3})$$

Table 7 compares the effect in terms of both Time and Space of analyzing the warming-up example for different sizes of $|\mathcal{F}|$ and $|Q|$. We report only the performance results for the property φ_1, since we obtain similar results for φ_2. We observe that the occupied memory Space grows exponentially with the number of features $|\mathcal{F}|$ and quantifiers $|Q|$ (when $|Q| = |\mathcal{F}|$), thus representing the bottleneck of the verification task. In fact, the size of the explored model spaces increases very rapidly with the size of the tuples, making the reasoning on the models very prohibitive. Note that the size of tuples is identical to the number of quantifiers in the given property. Figure 8 (left) depicts this phenomenon. It shows the occupied memory (in MB) of using ProVeLines to verify property φ_1 for increasing number of features and quantifiers, when $|Q| = |\mathcal{F}|$. Figure 8 (right) shows the accumulated time (in sec) for increasing number of features and quantifiers, when $|Q| = |\mathcal{F}|$. We can see that the time also grows exponentially with $|\mathcal{F}|$ and $|Q|$. On the other hand, if the number of quantifiers $|Q|$ is fixed ($|Q| = 2, 4$, or 6), we observe only linear growth of Time and Space for increasing number of features $|\mathcal{F}|$. This is due to the fact that we work with the same sized tuples in those cases and the ProVeLines tool can efficiently handle the variability in this simple example.

6.3 MinePump example

The MinePump system was introduced in the CONIC project [32]. Based on the original system, an fPromela model was created in [10] as part of the textscSNIP project. The fPromela MinePump family contains about 200 LOC and 7 (non-mandatory) independent optional features: `Start`, `Stop`, `MethaneAlarm`, `MethaneQuery`, `Low`, `Normal`, and `High`, thus yielding $2^7 = 128$ variants. Its FTS has 21,177 states and all variants combined have 889,252 states. It consists of 5 communicating processes: a `controller`, a `pump`, a `watersensor`, a `methanesensor`, and a `user`. When activated, the controller should switch on the pump when the water level is high, but only if there is no methane in the mine.

6.4 Discussion

Our proof-of-concept model checker for the alternation-free fragment of fMultiLTL is limited to smaller system families as evidenced by experiments. It represents a demonstration that model checking of fMultiLTL properties is possible. In the future work, we aim to propose some optimization heuristics that will reduce the computational complexity of model checking both fMultiLTL$_1$ in practice, and thus enable us to handle bigger real-world case studies. We also envision to leverage modern verification techniques like IC3 [36], interpolation [33], SMT [17] to improve the current algorithms on model checking of fMultiLTL.

7 RELATED WORK

In the last two decades, researchers have introduced various family-based (lifted) analysis and verification techniques for SPLs. Some successful examples range from family-based syntax and type checking [26, 30, 31], to family-based static analysis [3, 19–21, 40] and family-based verification by simulation [29, 41]. Family-based model checking has also been an active research field, where different approaches have been developed for verifying system families. Among various modelling formalisms for representing SPLs, we focus here on FTSs. We divide our discussion of related work into two categories: family-based model checking on FTSs and temporal logics for hyper- and multi-properties.

Family-based model checking on FTSs. Featured transition systems (FTSs) are today widely accepted formalism for representing system families (SPLs). Specialized family-based model checking

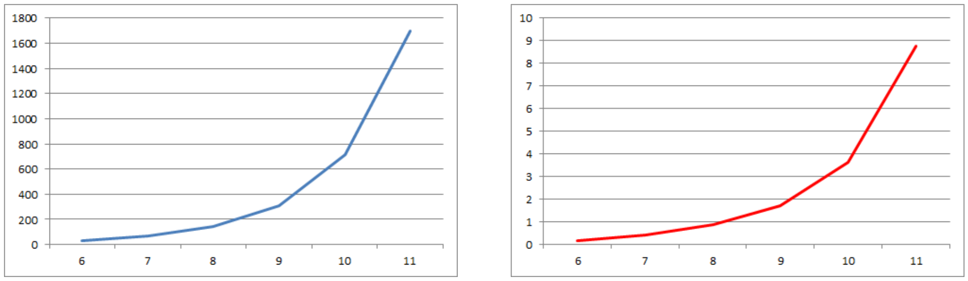

Figure 8: The performance of family-based model checking with ProVeLines as a function of the number of features $|\mathcal{F}|$ and quantifiers $|Q|$ (when $|Q| = |\mathcal{F}|$). The x-axis represents the number of features and quantifiers, and the y-axis represents the occupied memory in MB (left) and verification time in seconds (right).

algorithms have been designed for verifying FTSs against LTL properties [10]. They have been implemented in the SNIP family-based model checker [9] and its successor ProVeLines [14]. Cordy et. al [16] have also introduced symbolic family-based model checking algorithms for verifying FTSs against CTL properties, which has been implemented as an extension of the NuSMV model checker. Family-based model checking has been also defined for verifying μ-calculus properties using the general-purpose mCRL2 model checker [37], whereas family-based model checking for verifying probabilistic system families has been defined in [5] and implemented in the ProFeat tool. To make all these algorithms based on FTSs more scalable, various abstractions have been applied. The so-called variability abstractions and the automatic abstraction-refinement procedures for efficient family-based model checking of LTL are proposed in [18, 24]. Subsequently, the above procedures have been extended for verifying CTL and μ-calculus properties [22]. Abstraction-refinement procedures for family-based model checking have also been proposed for LTL properties of reactive system families [15] and reachability properties of probabilistic system families [4]. In this paper, we pursue this line of work by proposing specifically designed family-based model checking algorithms for verifying fMultiLTL properties of FTSs.

Temporal logics for hyper- and multi-properties. Hyper-properties [7] represent a formalism for specifying properties of sets of traces, by quantification over traces in the system. They are especially suitable for specifying security properties, such as secure information flow and non-interference. The logic HyperLTL and HyperCTL* have been introduced in [6]. This work also proposes one of the first algorithms for model checking hyper-properties by combining self-composition and the classical LTL model checking. Self-composition combines several disjoint copies of the same system, allowing to express relationships among multiple traces. Subsequently, more scalable approach has been defined using alternating Büchi automaton [25]. The notion of hyper-properties is generalized to multi-properties in [27], which describes the behaviour of not just a single system, but of a set of systems called multi-model. While hyper-properties relate traces from the same system, multi-properties relate traces from the different components in the multi-model. Goudsmid et. al [27] introduce direct algorithms for model checking multi-properties from the MultiLTL logic. In this work, we further generalize the notion of multi-properties to fMultiLTL logic, which explicitly relates traces from the various sub-families of a system family (SPL).

8 CONCLUSION

In this work, we proposed a new fMultiLTL logic for specifying multi-properties of system families. We have described two algorithms for model checking of fMultiLTL_1 and fMultiLTL_2 fragments of the new logic. An implementation in ProVeLines is suggested applicable to quantifier-free fMultiLTL properties. The evaluation confirms that some interesting properties can be efficiently verified in this way. However, it also establishes that reasoning on the self-composed products is computationally very demanding.

As a future work, we plan to employ abstraction-based techniques [18, 24] to avoid the construction of the full product. We can use abstractions to compute approximations of all sub-families represented in the full product, such that if model checking the abstract full product is successful, we conclude that model checking the original full product holds. Since the abstract sub-families are much smaller models than the original ones, we can use this technique for accelerating model checking of multi-properties.

REFERENCES

[1] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking.* MIT Press.
[2] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 2004. Secure Information Flow by Self-Composition. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004).* IEEE Computer Society, 100–114. https://doi.org/10.1109/CSFW.2004.17
[3] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. 2013. SPL^{LIFT}: statically analyzing software product lines in minutes instead of years. In *ACM SIGPLAN Conference on PLDI '13.* 355–364.
[4] Milan Ceska, Christian Dehnert, Nils Jansen, Sebastian Junges, and Joost-Pieter Katoen. 2019. Model Repair Revamped: On the Automated Synthesis of Markov Chains. In *Essays Dedicated to Scott A. Smolka on the Occasion of His 65th Birthday (LNCS, Vol. 11500).* Springer, 107–125. https://doi.org/10.1007/978-3-030-31514-6_7
[5] 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. https://doi.org/10.1007/s00165-017-0432-4
[6] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. 2014. Temporal Logics for Hyperproperties. In *Principles of Security and Trust - Third International Conference, POST 2014, Proceedings (LNCS, Vol. 8414).* Springer, 265–284. https://doi.org/10.1007/978-3-642-54792-8_15
[7] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *J. Comput. Secur.* 18, 6 (2010), 1157–1210. https://doi.org/10.3233/JCS-2009-0393

[8] Andreas Classen, Quentin Boucher, and Patrick Heymans. 2011. A text-based approach to feature modelling: Syntax and semantics of TVL. *Sci. Comput. Program.* 76, 12 (2011), 1130–1143. https://doi.org/10.1016/j.scico.2010.10.005

[9] Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. 2012. Model checking software product lines with SNIP. *STTT* 14, 5 (2012), 589–612. https://doi.org/10.1007/s10009-012-0234-1

[10] 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

[11] 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

[12] 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*. ACM, 335–344. https://doi.org/10.1145/1806799.1806850

[13] Paul Clements and Linda Northrop. 2001. *Software Product Lines: Practices and Patterns*. Addison-Wesley.

[14] Maxime Cordy, Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. 2013. ProVeLines: a product line of verifiers for software product lines. In *17th International Software Product Line Conference co-located workshops, SPLC 2013 workshops*. ACM, 141–146. https://doi.org/10.1145/2499777.2499781

[15] Maxime Cordy, Patrick Heymans, Axel Legay, Pierre-Yves Schobbens, Bruno Dawagne, and Martin Leucker. 2014. Counterexample guided abstraction refinement of product-line behavioural models. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22)*. 190–201. https://doi.org/10.1145/2635868.2635919

[16] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. 2013. Beyond boolean product-line model checking: dealing with feature attributes and multi-features. In *35th International Conference on Software Engineering, ICSE '13*. IEEE Computer Society, 472–481. https://doi.org/10.1109/ICSE.2013.6606593

[17] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008. Proceedings (LNCS, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[18] Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wasowski. 2016. Efficient family-based model checking via variability abstractions. *STTT* (2016). https://doi.org/10.1007/s10009-016-0425-2

[19] Aleksandar S. Dimovski and Sven Apel. 2021. Lifted Static Analysis of Dynamic Program Families by Abstract Interpretation. In *35th European Conference on Object-Oriented Programming, ECOOP 2021 (LIPIcs, Vol. 194)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 14:1–14:28. https://doi.org/10.4230/LIPIcs.ECOOP.2021.14

[20] Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. 2015. Variability Abstractions: Trading Precision for Speed in Family-Based Analyses. In *29th European Conference on Object-Oriented Programming, ECOOP 2015 (LIPIcs, Vol. 37)*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 247–270. https://doi.org/10.4230/LIPIcs.ECOOP.2015.247

[21] Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. 2019. Finding suitable variability abstractions for lifted analysis. *Formal Aspects Comput.* 31, 2 (2019), 231–259. https://doi.org/10.1007/s00165-019-00479-y

[22] 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. https://doi.org/10.1016/j.tcs.2020.06.011

[23] Aleksandar S. Dimovski and Andrzej Wasowski. 2017. From Transition Systems to Variability Models and from Lifted Model Checking Back to UPPAAL. In *Models, Algorithms, Logics and Tools - Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday (LNCS, Vol. 10460)*. Springer, 249–268. https://doi.org/10.1007/978-3-319-63121-9_13

[24] Aleksandar S. Dimovski and Andrzej Wasowski. 2017. Variability-specific Abstraction Refinement for Family-Based Model Checking. In *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Proceedings (LNCS, Vol. 10202)*. 406–423. https://doi.org/10.1007/978-3-662-54494-5_24

[25] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. 2015. Algorithms for Model Checking HyperLTL and HyperCTL ˆ*. In *Computer Aided Verification - 27th International Conference, CAV 2015, Proceedings, Part I (LNCS, Vol. 9206)*. Springer, 30–48. https://doi.org/10.1007/978-3-319-21690-4_3

[26] Paul Gazzillo and Robert Grimm. 2012. SuperC: parsing all of C by taming the preprocessor. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 323–334. https://doi.org/10.1145/2254064.2254103

[27] 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, Proceedings (LNCS, Vol. 12597)*. Springer, 55–80. https://doi.org/10.1007/978-3-030-67067-2_4

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

[29] Alexandru F. Iosif-Lazar, Jean Melo, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. 2017. Effective Analysis of C Programs by Rewriting Variability. *Programming Journal* 1, 1 (2017), 1. https://doi.org/10.22152/programming-journal.org/2017/1/1

[30] Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. 2012. Type checking annotation-based product lines. *ACM Trans. Softw. Eng. Methodol.* 21, 3 (2012), 14.

[31] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011*. 805–824. https://doi.org/10.1145/2048066.2048128

[32] Jeff Kramer, Jeff Magee, Morris Sloman, and A. Lister. 1983. CONIC: An Integrated Approach to Distributed Computer Control Systems. *IEE Proc.* 130, 1 (1983), 1–10.

[33] Kenneth L. McMillan. 2003. Craig Interpolation and Reachability Analysis. In *Static Analysis, 10th International Symposium, SAS 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2694)*. Springer, 336. https://doi.org/10.1007/3-540-44898-5_18

[34] Klaus Pohl, Günter Böckle, and Frank van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer.

[35] Riccardo Pucella and Fred B. Schneider. 2006. Independence From Obfuscation: A Semantic Framework for Dive. In *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006)*. IEEE Computer Society, 230–241. https://doi.org/10.1109/CSFW.2006.15

[36] Fabio Somenzi and Aaron R. Bradley. 2011. IC3: where monolithic and incremental meet. In *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11*. FMCAD Inc., 3–8. http://dl.acm.org/citation.cfm?id=2157657

[37] Maurice H. ter Beek, Erik P. de Vink, and Tim A. C. Willemse. 2017. Family-Based Model Checking with mCRL2. In *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Proceedings (LNCS, Vol. 10202)*. 387–405. https://doi.org/10.1007/978-3-662-54494-5_23

[38] Tachio Terauchi and Alexander Aiken. 2005. Secure Information Flow as a Safety Problem. In *Static Analysis, 12th International Symposium, SAS 2005, Proceedings (LNCS, Vol. 3672)*, Chris Hankin and Igor Siveroni (Eds.). Springer, 352–367. https://doi.org/10.1007/11547662_24

[39] Moshe Y. Vardi and Pierre Wolper. 1986. An Automata-Theoretic Approach to Automatic Program Verification. In *Proceedings of the Symposium on Logic in Computer Science (LICS '86)*. IEEE Computer Society, 332–344.

[40] Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven 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

[41] Alexander von Rhein, Thomas Thüm, Ina Schaefer, Jörg Liebig, and Sven Apel. 2016. Variability encoding: From compile-time to load-time variability. *J. Log. Algebr. Meth. Program.* 85, 1 (2016), 125–145. https://doi.org/10.1016/j.jlamp.2015.06.007