**Software Tools for Technology Transfer manuscript No.**
(will be inserted by the editor)

# Efficient family-based model checking via variability abstractions ⋆

**Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, Andrzej Wąsowski**

IT University of Copenhagen, Denmark

Received: date / Revised version: date

**Abstract.** Many software systems are variational: they can be configured to meet diverse sets of requirements. They can produce a (potentially huge) number of related systems, known as products or variants, by systematically reusing common parts. For variational models (variational systems or families of related systems), specialized *family-based* model checking algorithms allow efficient verification of multiple variants, simultaneously, in a single run. These algorithms, implemented in a tool SNIP, scale much better than "the brute force" approach, where all individual systems are verified using a single-system model checker, one-by-one. Nevertheless, their computational cost still greatly depends on the number of features and variants. For variational models with a large number of features and variants, the family-based model checking may be too costly or even infeasible.

In this work, we address two key problems of family-based model checking. First, we improve scalability by introducing abstractions that simplify variability. Second, we reduce the burden of maintaining specialized family-based model checkers, by showing how the presented variability abstractions can be used to model check variational models using the standard version of (single-system) SPIN. The variability abstractions are first defined as Galois connections on semantic domains. We then show how to use them for defining abstract family-based model checking, where a variability model is replaced with an abstract version of it, which preserves the satisfaction of LTL properties. Moreover, given an abstraction, we define a syntactic source-to-source transformation on high-level modelling languages that describe variational models, such that the model checking of the transformed high-level variational model coincides with the abstract model checking of the concrete high-level variational model. This allows the use of SPIN with all its accumulated optimizations for efficient verification of variational models without any knowledge about variability. We have implemented the transformations in a prototype tool, and we illustrate the practicality of this method on several case studies.

⋆ Danish Council for Independent Research, Sapere Aude grant no. 0602-02327B

## 1 Introduction

Variability is an increasingly frequent phenomenon in software systems. A growing number of projects adopt the *Software Product Line* (SPL) methodology [15] for building a *family* of related systems. Implementations of such systems usually [1] contain statically configured options (variation points, features) governed by a *feature configuration*. A feature configuration determines a single *variant* (product) of the family, which can be derived, built, tested, and deployed. The SPL methodology is particularly popular in the embedded systems domain, where organizing development and production in product lines is very common (e.g., cars, phones) [15].

Variability plays a significant role outside of the SPL methodology as well. Many communication protocols, components and system-level programs are highly configurable: a set of parameters is decided/implemented statically and then never changes during execution. These systems interpret decisions over variation point at runtime, instead of statically configuring them. Nevertheless, since the configurations do not normally change during the time of execution, the abstract semantics of load-time variational systems is similar to static SPLs. Therefore, variational systems, i.e. families of related systems, can be conceptually specified using *variational* models. The semantics of variational models is specified in two stages: first the selection of features is decided and the model is specialized (preprocessed) for a given configuration. Afterwards, it is interpreted using the standard non-variational semantics.

Since embedded systems, system-level software and communication protocols frequently serve critical functions, they require rigorous validation of models. *Model checking* is a well-known technique for automatic verification of system designs: a model of the system is constructed and the requirements represented as temporal logic formulae are checked over this model. Performance of classical single-variant model checking algorithms depends on the size of the model and the size of the specification property [3]. Classical model checking research provides abstraction and reduction techniques to address the complexity stemming from both the model and the specification [9,37,20,28,27]. In most of these works, the generation of the abstract model is based on abstract interpretation theory [18]: the semantics of the concrete model is related with the semantics of its abstract version by using Galois connections. Provided that the abstraction preserves the property we want to check, the analysis of the smaller abstract model suffices to decide the satisfaction of the property on the concrete model. This topic is known as *abstract model checking*.

Unfortunately, model checking families of systems is harder than model checking single systems because, combinatorically, the number of possible variants is exponential in the number of features (configuration parameters). Hence, the "brute force" approach, that applies *single-system* model checking to each individual variant of a *family-based* system, one-by-one, is inefficient. To circumvent this problem, *family-based model checking* (also known as lifted model checking) algorithms have been proposed [11,12,14], implemented in the tool, $\overline{\text{SNIP}}$.[1] These algorithms model check *all* variants of a family simultanously and pinpoint the variants that violate properties. However, efficiency of these algorithms *still* depend on the size of the configuration space, which is still inherently exponential in the number of configuration parameters. In order to handle variational models efficiently we need abstraction and reduction techniques that address the size of the configuration space, not only the size of the model and of the specification.

### 1.1 Overview of our technique

In this paper, we use abstract interpretation to define a calculus of property preserving *variability abstractions* for variational models. Thus, we lay the foundations for *abstract* family-based model checking of variational models.

Fig. 1 illustrates our technique. The top left corner shows an *f*PROMELA program (model) that describes a given variational system. *f*PROMELA is a high-level modelling language—a variability-aware extension of

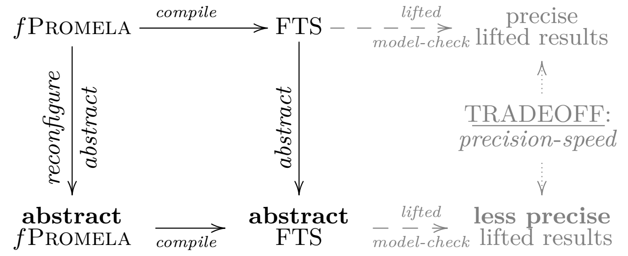



Fig. 1: An overview of our technique for efficient family-based model checking. Instead of abstracting the semantics of a concrete variational model, our transformation allows abstraction to be applied directly to the source code of the variational model.

PROMELA, the language of SPIN model checker [30]. It adds feature variables and a new guarded-by-features statement to PROMELA. These allow to make selected statements available only to a subset of the variants. *f*PROMELA models are compiled to the so-called *Featured Transition Systems* (FTSs) [11,12]. See mid-top in Fig. 1. FTSs extend the regular transitions systems (TSs) [3] by enriching transitions with a guard predicate over features that specifies for which variants the transition is enabled. We can run a lifted, family-based [44], model checking algorithms directly over an FTS and obtain "*precise* lifted results" (top-right in the figure). This means that for each variant we obtain an answer whether or not it satisfies a given property, and we also obtain a counter-example for each violation. Since running the lifted model checking might be too slow or infeasible, we may decide to use *abstraction* to obtain a faster, although less precise verification results.

Classically, an abstraction is applied to FTSs (middle downward arrow in the figure), which produces an "abstract FTS" with much smaller size than the original FTS. When lifted model checking algorithms are subsequently run, they will produce "less precise verification results", but it will do so faster than the original analysis (i.e., there is a *precision vs. speed tradeoff*). The obtained results are less precise in the sense that some of the counter-examples reported may be spurious—introduced in the model during abstraction.

Interestingly, for model checking of *f*PROMELA, *abstraction* (downward arrow) and *compilation* (rightward arrow) commute which means that we may swap their order of application. The implications are significant. It means that variability abstractions can be applied *before* compilation; i.e., directly on *f*PROMELA models. We exploit this observation to define a source-to-source transformation, called RECONFIGURATOR, on the source level of input models, which enable an effective computation of abstract models syntactically from high-level modelling languages. The two paths from a variational system to

[1] The project on development of $\overline{\text{SNIP}}$ model checker (https://projects.info.unamur.be/fts/) is independent of SPIN. $\overline{\text{SNIP}}$ has been implemented from scratch. We put a line over $\overline{\text{SNIP}}$ to make the distinction from SPIN clearer.

"abstract lifted model checking" are guaranteed to produce the same abstract lifted results. This makes our technique easier to implement than using the semantic-based abstractions defined for FTSs. In this way, we avoid the need for intermediate storage in memory of the semantics of the concrete full-blown variational model. It also opens up a possibility of verifying properties of variational models, so of multiple variants simultanously, without using a dedicated family-based model checker such as $\overline{\text{SNIP}}$. We can use variability abstraction to obtain an abstracted family-of-models (with a low number of variants) that can then be model checked via "brute force" using a single-system model checker, such as SPIN [30].

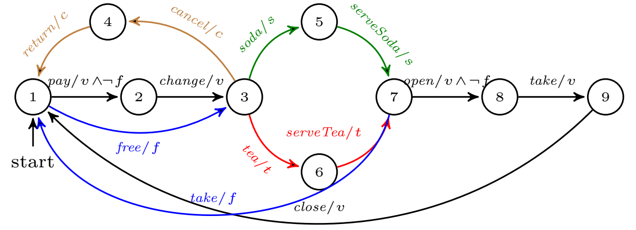



Fig. 2: VendingMachine family-of-models with five features, each assigned an identifying letter and a color: $v$ for `VendingMachine` (in black), $t$ for `Tea` (in red), $s$ for `Soda` (in green), $c$ for `CancelPurchase` (in brown), and $f$ for `FreeDrinks` (in blue).

### 1.2 Contribution and organization

We make the following contributions:

- **Variability abstractions.** A class of variability abstractions for featured transition systems, defined inductively using a simple compositional calculus of Galois connections.
- **Soundness of abstraction.** A soundness result for the proposed abstractions, with respect to LTL properties.
- **Abstraction via syntactic transformation.** A definition of the abstraction operators as source-to-source transformations on variational models. The transformations are shown to have the same effect as applying the abstractions to the semantics of models (featured transitions systems). This allows us to apply the abstractions in a preprocessing step, without any modifications to the model checking tools.
- **Efficient family-based model checking.** A technique for performing family-based model checking using an off-the-shelf model checker. This technique relies on partitioning and abstracting the variational models until the point when they contain no variability. The default highly optimized implementation of SPIN can then be used to verify the resulting abstracted models.
- **Experimental evaluation.** An experimental evaluation exploring the effectiveness of the above method of family-based model checking with SPIN, as well as the impact of abstractions on the scalability of the state of the art family-based model checker $\overline{\text{SNIP}}$.

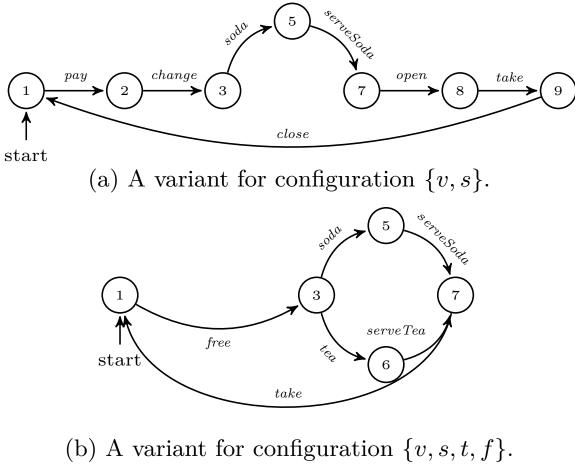
This paper targets researchers and practitioners who already use model checking in their projects, but, so far, have only been analyzing one variant at a time, as well as those working on family-based model checking. Although designed for SPIN, the proposed rewrite techniques shall be easily extensible to other model checkers, including probabilistic and real-time models. Also the designers of efficient family-based model checkers may find the methodology of applying abstractions ahead of analysis interesting, as it is much more lightweight to implement, yet very effective, as shown in our experiments.

This work represents an extended and revised version of [23]. Compared to the earlier work, we make the following extensions here. We motivate the need for using variability abstractions for achieving efficient family-based model checking. We provide formal proofs for all main results. We expand and elaborate the examples. Finally, we provide more evaluation results as well as one additional case study.

We proceed by giving a motivating example for applying variability abstractions to lifted model checking in Section 2. The basics of lifted model checking are explained in Section 3. Section 4 defines a calculus for specification of variability abstractions. Section 5 explains how to use high-level modelling languages to encode families of related systems. The Reconfigurator transformation is described in Section 6 along with the correctness result. Section 7 presents the evaluation on two case studies. In Section 8, we show how our verification procedure can be converted into a fully automatic. Finally, we discuss the relation to other works and conclude.

## 2 Motivating Example

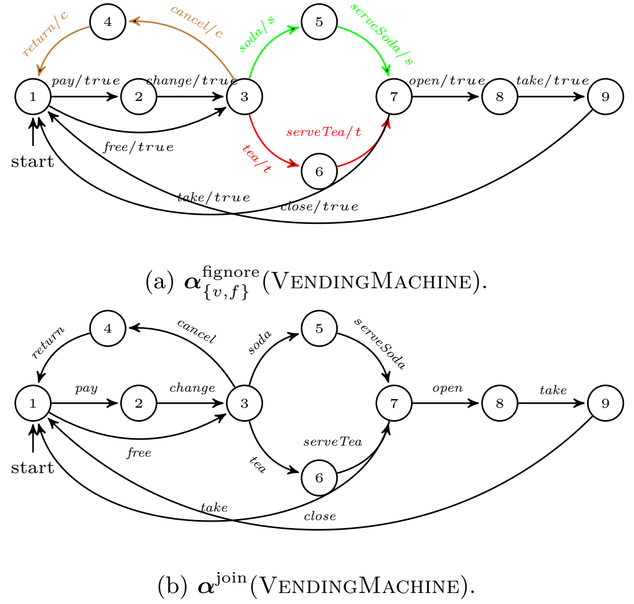
To better illustrate the issues we are addressing in this work, we now present a motivating example. Fig. 2 shows the FTS of a VendingMachine [12] which is a variational system that describes the behaviors of a family of models of vending machines. The VendingMachine family has five features (the set denoted $\mathbb{F}$). Each of them is assigned an identifying letter and a color: `VendingMachine` (denoted by $v$, in black), for purchasing a drink which represents a mandatory root feature enabled in all variants; `Tea` ($t$, in red), for serving tea; `Soda` ($s$, in green), for serving soda; `CancelPurchase` ($c$, in brown), for canceling a purchase after a coin is entered; and `FreeDrinks` ($f$, in blue) for offering free drinks. Each transition is labeled by first an *action*, then a slash '/', and finally a guarding *feature expression* denoting whether or not the transition is to be included in a given variant, depending on which features have been enabled. For readability, we color the transitions according to the feature expression guarding it. For instance, the transition ③ $\xrightarrow{soda/s}$ ⑤ is



(a) A variant for configuration $\{v, s\}$.

(b) A variant for configuration $\{v, s, t, f\}$.

Fig. 3: Some variants of VendingMachine.



(a) $\boldsymbol{\alpha}^{\text{fignore}}_{\{v,f\}}$(VendingMachine).

(b) $\boldsymbol{\alpha}^{\text{join}}$(VendingMachine).

Fig. 4: Some abstractions of VendingMachine.

guarded by feature expression, $s$ (in green), which means that it is only included in models if the (green) feature `Soda` has been enabled.

A number of variants of this VendingMachine can be derived. Fig. 3a shows the basic variant of VendingMachine that only serves soda, where only the features $v$ (black) and $s$ (green) have been enabled. It accepts payment, returns change, serves a soda, opens the access compartment, so that the customer can take the soda, and closes it again. A second variant of this machine, shown in Fig. 3b, extends the basic one by serving tea and offering free drinks, instead of requiring payment. The following features have been enabled: $v$ (black), $s$ (green), $t$ (red), and $f$ (blue). It shows the impact of adding features $t$ and $f$ to the basic variant. `Tea` ($t$) adds two transitions: ③ $\xrightarrow{tea}$ ⑥ and ⑥ $\xrightarrow{serveTea}$ ⑦; whereas `FreeDrinks` ($f$) adds: ① $\xrightarrow{free}$ ③ and ⑦ $\xrightarrow{take}$ ①, but removes: ① $\xrightarrow{pay}$ ② and ⑦ $\xrightarrow{open}$ ⑧. Note that transitions ② $\xrightarrow{change}$ ③ and ⑧ $\xrightarrow{take}$ ⑨ $\xrightarrow{close}$ ① are not present in this variant, since they are not reachable from the initial state ①.

Other variants of the VendingMachine can be also generated by selecting (enabling, respectively, disabling) other combinations of features. Combinatorically, this gives rise to $2^{|\mathbb{F}|}$ different variants (configurations). In general, however, not all combinations of features are *valid*. Obviously, the basic vending machine functionality feature, $v$ (black), must be enabled in all variants. Equally clearly, any variant of VendingMachine should serve at least one kind of drink (either *soda* or *tea*). Therefore, any *valid* variant of our VendingMachine should have mandatory feature $v$ (black) enabled and $s$ (green) or $t$ (red) enabled, possibly both.

Let us suppose that some proposition, `select`, holds in state three ③. Now consider an example property, $P$, which states that *"`select` holds infinitely often"*; i.e.:

> *"infinitely often, a customer is able to `select` between taking a drink or canceling a purchase."*

Both variants of VendingMachine in Fig. 3 satisfy this property, since the state ③ is reachable infinitely often in any execution of those variants. In fact, all variants of VendingMachine satisfy $P$. However, in order to formally verify $P$, we can either instantiate all exponentially many valid variants from VendingMachine and verify them, one-by-one (known as "the brute-force approach"); or, we can apply recently invented *family-based* model checking algorithms [11,12,14] (implemented in the $\overline{\text{SNIP}}$ tool) that work on the *family-based* level of FTSs rather than individual *single-system* TSs. Although the family-based approach is more efficient, it still inherently depends on the number of features and variants in the family.

In order to speed up the above family-based verification procedure, we introduce so-called variability abstractions which may tame the exponential explosion of the number of variants and reduce it to something more tractable. In effect, we obtain a computationally cheaper but less precise verification procedure, since an over-approximation is introduced in it. The two basic variability abstractions in our calculus are: (1) to confound (join) all valid variants into one system, denoted $\boldsymbol{\alpha}^{\text{join}}$, and (2) to ignore a set of features, $\mathbb{A} \subseteq \mathbb{F}$, deemed as not relevant for the current problem, denoted $\boldsymbol{\alpha}^{\text{fignore}}_{\mathbb{A}}$. With $\boldsymbol{\alpha}^{\text{join}}$ we obtain a single system with over-approximated control flow, whereas with $\boldsymbol{\alpha}^{\text{fignore}}_{\mathbb{A}}$ we obtain a family with less variability by confounding all executions that differ only with regard to the features in $\mathbb{A}$. We also use a projection (divide-and-conquer) operator, which partitions the space of all variants into a number of subsets

that can be analyzed one at a time, and a composition operator to build other more sophisticated verification strategies out of these basic operators.

Notice that satisfaction of the property $P$ in VendingMachine does not depend on whether the features $v$ and $f$ are enabled. So, we can ignore them by replacing in feature expressions all literals corresponding to $v$ and $f$ with *true*. We still keep precision with respect to all other features: $c$, $t$ and $s$. This will result in a new FTS, $\boldsymbol{\alpha}^{\text{fignore}}_{\{v,f\}}$(VendingMachine), given in Fig. 4a. Compared to the concrete VendingMachine in Fig. 2, the new FTS has less variability (less number of features and variants) and more executions (over-approximation). For example, the execution in which the machine first asks a customer for a coin and then offers a free drink to the next customer is possible in $\boldsymbol{\alpha}^{\text{fignore}}_{\{v,f\}}$(VendingMachine), but not in VendingMachine. This is so, because the transitions ① $\xrightarrow{free}$ ③ and ① $\xrightarrow{pay}$ ② $\xrightarrow{change}$ ③ are alternatives in VendingMachine, and their presence in a variant depends on whether the feature $f$ is enabled or not. In effect, the new abstract verification procedure runs faster reporting that $P$ holds for $\boldsymbol{\alpha}^{\text{fignore}}_{\{v,f\}}$(VendingMachine), which implies that the property $P$ holds for the concrete VendingMachine as well.

Since the above property $P$ does not depend on the presence of any feature, we can apply the coarsest abstraction $\boldsymbol{\alpha}^{\text{join}}$, which simply confounds (joins) control flows of all valid variants into a single system, where all feature expressions hold (they become *true*). As result of doing this we obtain $\boldsymbol{\alpha}^{\text{join}}$(VendingMachine), shown in Fig. 4b, which over-approximates the concrete VendingMachine. Also, $\boldsymbol{\alpha}^{\text{join}}$(VendingMachine) has no variability and represents an ordinary transition system. Hence, it can be verified efficiently using classical (single-system) model checking algorithms (implemented in tools such as SPIN). The use of $\overline{\text{SNIP}}$ is no longer needed in this case. By verifying that $P$ holds for $\boldsymbol{\alpha}^{\text{join}}$(VendingMachine), we can conclude that it also holds for VendingMachine.

The motivating example we considered here, VendingMachine, was phrased as an FTS (cf. Fig. 2). In practice, however, models are usually not phrased directly as low-level FTSs. Instead, higher-level languages are used; in particular, the language *f*Promela, for specifying families of models.

As mentioned in the introduction, we observe that *variability abstraction* (the downward arrow of Fig. 1) and *model compilation* (right arrow) commute. We exploit this to make all *variability abstractions*, as presented above on FTSs, work on the level of *f*Promela in the form of the source-to-source Reconfigurator tool. Given a variability abstraction, our Reconfigurator is able to transform an *f*Promela model-family into an *abstracted* *f*Promela model-family that behaves exactly as if the original model-family had been first compiled and only then abstracted and analyzed. The result is that we may combine our source-to-source Reconfigurator with existing model checkers *black-box* and effectively augment them with abstraction, irrespective of their internal behavior, representations, and optimizations.

## 3 Modelling Variational Behavior

A common way of introducing variability into modeling languages is superimposing multiple variants in a single model [19]. Following this, Classen et al. present *featured transition systems* [12,14], an extension of transition systems [3] with static guard conditions over features on transitions. The guards determine in which variants the transitions appear. An FTS represents the behaviour of all instances of a variational system. Hence, all variants are represented in a single model in order to exploit the similarities between them. The set of valid configurations is encoded in a separate *feature model* [31], i.e. a tree-like structure that specifies which combinations of features are valid. They have also proposed specifically designed family-based model checking algorithms for verification of FTSs against LTL properties and implemented them in the $\overline{\text{SNIP}}$ tool [11].

### *3.1 Featured Transition Systems*

Let $\mathbb{F} = \{A_1, \dots, A_n\}$ be a finite set of Boolean variables representing the features available in a variational model. A *configuration* is a specific subset of features $k \subseteq \mathbb{F}$. Each configuration defines a variant of a model. Only a subset $\mathbb{K} \subseteq 2^{\mathbb{F}}$ of configurations are valid. Equivalently, configurations can be represented as formulae (minterms). Each configuration $k \in \mathbb{K}$ can be represented by the term $\bigwedge_{i=1..n} \nu(A_i)$ where $\nu(A_i) = A_i$ if $A_i \in k$, and $\nu(A_i) = \neg A_i$ if $A_i \notin k$. Since minterms can be bijectively translated into sets of features, we use both representations interchangeably. The set of valid configurations is typically described by a feature model, but we disregard syntactic representations of the set $\mathbb{K}$ in this paper.

The behaviour of individual variants is given as transition systems.

**Definition 1.** A tuple $\mathcal{T} = (S, Act, trans, I, AP, L)$ is a transition system, where $S$ is a set of states, $Act$ is a set of actions, $trans \subseteq S \times Act \times S$ is a transition relation, $I \subseteq S$ is a set of initial states, $AP$ is a set of atomic propositions, and $L : S \to 2^{AP}$ is a labeling function. We write $s_1 \xrightarrow{\lambda} s_2$ when $(s_1, \lambda, s_2) \in trans$.

An *execution* (behaviour) of a transition system $\mathcal{T}$ is a nonempty, infinite sequence $\rho = s_0\lambda_1 s_1 \lambda_2 \dots$ such that $s_0 \in I$ and $s_i \xrightarrow{\lambda_{i+1}} s_{i+1}$ for all $i \geq 0$. The *semantics* of $\mathcal{T}$, written $[\![\mathcal{T}]\!]_{\text{TS}}$, is the set of all executions of $\mathcal{T}$.

Let $FeatExp(\mathbb{F})$ denote the set of all propositional formulae over $\mathbb{F}$ generated using the grammar:

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

For a condition $\psi \in \mathit{FeatExp}(\mathbb{F})$, we write $[\![\psi]\!]$ to mean the set of valid variants that satisfy $\psi$, i.e. $k \in [\![\psi]\!]$ iff $k \models \psi$ and $k \in \mathbb{K}$, where $\models$ denotes the standard satisfaction of propositional logic.

The combined behaviour of a whole system family is compactly represented with FTSs. They are basically TSs appropriately decorated with feature expressions.

**Definition 2.** A tuple $\mathcal{F}=(S, \mathit{Act}, \mathit{trans}, I, AP, L, \mathbb{F}, \mathbb{K}, \delta)$ is a featured transition system if $(S, \mathit{Act}, \mathit{trans}, I, AP, L)$ is a transition system, $\mathbb{F}$ is the set of available features, $\mathbb{K}$ is a set of valid configurations, and $\delta : \mathit{trans} \to \mathit{FeatExp}(\mathbb{F})$ is a total function labeling transitions with feature expressions.

The *projection* of an FTS $\mathcal{F}$ onto a variant $k \in \mathbb{K}$, written $\pi_k(\mathcal{F})$, is a transition system $(S, \mathit{Act}, \mathit{trans}', I, AP, L)$, where $\mathit{trans}' = \{t \in \mathit{trans} \mid k \models \delta(t)\}$. Projection is analogous to *preprocessing* of `#ifdef` statements in C/CPP family-based SPLs and is naturally lifted to *sets* of variants. Given $\mathbb{K}' \subseteq \mathbb{K}$, the projection $\pi_{\mathbb{K}'}(\mathcal{F})$ is the FTS $(S, \mathit{Act}, \mathit{trans}', I, AP, L, \mathbb{F}, \mathbb{K}', \delta)$, where $\mathit{trans}' = \{t \in \mathit{trans} \mid \exists k \in \mathbb{K}'.k \models \delta(t)\}$. The *semantics* of the FTS $\mathcal{F}$, written $[\![\mathcal{F}]\!]_{\mathrm{FTS}}$, is the union of the behavior of the projections onto all valid variants $k \in \mathbb{K}$, i.e. we have:

$$[\![\mathcal{F}]\!]_{\mathrm{FTS}} = \bigcup_{k \in \mathbb{K}} [\![\pi_k(\mathcal{F})]\!]_{\mathrm{TS}}$$

Classen at el. [12] define the *size* of an FTS as: $|\mathcal{F}| = |S| + |\mathit{trans}| + |\mathit{expr}| + |\mathbb{K}|$, where $|\mathit{expr}|$ denotes the size of all feature expressions bounded by $O(2^{|\mathbb{F}|} \cdot |\mathit{trans}|)$. In these terms, our abstractions aim to reduce the $|\mathit{expr}|$ and $|\mathbb{K}|$ components of the size $|\mathcal{F}|$.

*Example 1.* Let us revisit the VendingMachine example from Section 2. Fig. 2 presents an FTS describing the behavior of the VendingMachine. It contains five features $\mathbb{F} = \{v, t, s, c, f\}$. We assume that only valid configurations are $\mathbb{K} = \{\{v, s\}, \{v, s, t, f\}, \{v, s, c\}, \{v, t, c, f\}\}$. The valid configuration $\{v, s\}$ corresponding to the basic variant, $\pi_{\{v,s\}}(\text{VendingMachine})$, given in Fig. 3a can be expressed as the formula $v \wedge s \wedge \neg t \wedge \neg c \wedge \neg f$. Note that the variant $\pi_{\{v,s,t,f\}}(\text{VendingMachine})$ is shown in Fig. 3b. □

### *3.2 fLTL Properties*

An LTL formula is defined inductively as:

$$\phi ::= \mathit{true} \mid a \in AP \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid X\phi \mid \phi_1 \, U \, \phi_2$$

Satisfaction of a formula $\phi$ for an infinite execution $\rho = s_0 \lambda_1 s_1 \lambda_2 \ldots$ (we write $\rho_i = s_i \lambda_{i+1} s_{i+1} \ldots$ for the $i$-th suffix of $\rho$) is defined as:

$$\begin{array}{ll}
\rho \models \mathit{true} & \text{always (for any } \rho) \\
\rho \models a & \text{iff } a \in L(s_0), \\
\rho \models \phi_1 \wedge \phi_2 & \text{iff } \rho \models \phi_1 \text{ and } \rho \models \phi_2, \\
\rho \models \neg\phi & \text{iff not } \rho \models \phi, \\
\rho \models \mathtt{X}\phi & \text{iff } \rho_1 \models \phi, \\
\rho \models \phi_1 \mathtt{U} \phi_2 & \text{iff } \exists k \geq 0 : \rho_k \models \phi_2 \text{ and} \\
 & \qquad \forall j. 0 \leq j < k-1 : \rho_j \models \phi_1
\end{array}$$

A TS $\mathcal{T}$ satisfies a formula $\phi$, written $\mathcal{T} \models \phi$, iff $\forall \rho \in [\![\mathcal{T}]\!]_{\mathrm{TS}} : \rho \models \phi$. Other temporal operators can be derived as usual: $\mathtt{F}\,\phi = \mathit{true}\,\mathtt{U}\,\phi$ (means "some Future state, eventually") and $\mathtt{G}\,\phi = \neg\mathtt{F}\neg\phi$ (means "Globally, always").

In the variational case, properties may hold only for some variants. To capture this in specifications, fLTL properties are quantified over the variants of interest.

**Definition 3.**
- A feature LTL (fLTL) formula is a pair $[\chi]\phi$, where $\phi$ is an LTL formula and $\chi$ is a feature expression from $\mathit{FeatExp}(\mathbb{F})$.
- An FTS $\mathcal{F}$ satisfies an fLTL formula $[\chi]\phi$, written $\mathcal{F} \models [\chi]\phi$, iff for all configurations $k \in \mathbb{K} \cap [\![\chi]\!]$ we have that $\pi_k(\mathcal{F}) \models \phi$. An FTS $\mathcal{F}$ satisfies an LTL formula $\phi$ iff $\mathcal{F} \models [\mathit{true}]\phi$.

*Example 2.* Consider the FTS for VendingMachine in Fig. 2. The property $P$: "`select` holds infinitely often" from Section 2 can be expressed as: "`G F select`". We can check that $P$ is indeed satisfied: VendingMachine $\models P$, since all valid variants (from $\mathbb{K}$) of VendingMachine satisfy $P$.

Suppose that states ⑤ and ⑥ are labeled `selected`, and the state ⑧ is labeled `open`. Consider an example property $\phi$ that after each time a beverage has been selected, the machine will eventually open the compartment to allow the customer to access his drink: `G(selected` $\implies$ `F open)`. The basic VendingMachine satisfies this property: $\pi_{\{v,s\}}(\text{VendingMachine}) \models \phi$, while the entire variational model does not, i.e. we have: VendingMachine $\not\models \phi$. For example, if the feature $f$ is enabled, the state ⑧ is unreachable, i.e. a counter-example (execution that violates $\phi$) is: ① → ③ → ⑤ → ⑦ → ① → .... The set of violating products is $\{\{v, s, t, f\}, \{v, t, c, f\}\} \subseteq \mathbb{K}$. At the same time, we can check that VendingMachine $\models [\neg f]\phi$. Therefore, we can conclude that the feature $f$ is responsible for violation of the property $\phi$. □

## 4 Variability Abstractions

We shall now introduce abstractions decreasing the sizes of FTSs, in particular the number of features $|\mathbb{F}|$ and the configuration space $|\mathbb{K}|$. We also show how these abstractions preserve fLTL properties over FTSs.

*4.1 A Calculus of Abstractions*

For fLTL model checking, variability abstractions can be defined over the set of features $\mathbb{F}$ and the configuration space $\mathbb{K}$, and then lifted to FTSs. This greatly simplifies the definitions. *Variability abstractions* aim to weaken feature expressions, in order to make transitions in FTSs available to more variants. We begin with the *complete Boolean lattice* of propositional formulae over $\mathbb{F}$: $(\mathit{FeatExp}(\mathbb{F})_{/\equiv}, \models, \vee, \wedge, \mathit{true}, \mathit{false})$. Elements of $\mathit{FeatExp}(\mathbb{F})_{/\equiv}$ are equivalence classes of propositional formulae obtained by quotienting by the semantic equivalence $\equiv$. The pre-order relation $\models$ is defined as the satisfaction relation from propositional logic, whereas the least upper bound operator is $\vee$ and the greatest lower bound operator is $\wedge$. Furthermore, the least element is $\mathit{false}$, and the greatest element is $\mathit{true}$ [2].

Sometimes the computational task on a *concrete* complete lattice (domain) may be too costly or even uncomputable and this motivates replacing it with a simpler *abstract* lattice. A *Galois connection* is a pair of total functions, $\alpha : L \to M$ and $\gamma : M \to L$ (respectively known as the *abstraction* and *concretization* functions), connecting two complete lattices, $\langle L, \leqslant_L\rangle$ and $\langle M, \leqslant_M\rangle$ (often called the *concrete* and *abstract* domain, respectively), such that:

$$\alpha(l) \leqslant_M m \iff l \leqslant_L \gamma(m) \text{ for all } l \in L, m \in M$$

which is often typeset as: $\langle L, \leqslant_L\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle M, \leqslant_M\rangle$. Note that $\leqslant_L$ and $\leqslant_M$ are the pre-order relations for $L$ and $M$, respectively.

*Join.* This abstraction confounds the control flow of all variants of the model, obtaining a single variant that includes all the executions occurring in any variant. The unreachable parts of the variational model that do not occur in any valid variant are eliminated. The information about which states belong to which variants is lost.

Technically, the abstraction collapses the entire configuration space onto a singleton set. Each feature expression $\psi$ is replaced with $\mathit{true}$ if $\psi$ is satisfied in at least one configuration from $\mathbb{K}$. The set of features in the abstracted model is empty: $\boldsymbol{\alpha}^{\text{join}}(\mathbb{F}) = \emptyset$, and the set of valid configurations is: $\boldsymbol{\alpha}^{\text{join}}(\mathbb{K}) = \{\mathit{true}\}$ if $\mathbb{K} \neq \emptyset$ and $\boldsymbol{\alpha}^{\text{join}}(\mathbb{K}) = \{\mathit{false}\}$ otherwise.
The *abstraction*, $\boldsymbol{\alpha}^{\text{join}} : \mathit{FeatExp}(\mathbb{F}) \to \mathit{FeatExp}(\emptyset)$, and *concretization* functions, $\boldsymbol{\gamma}^{\text{join}} : \mathit{FeatExp}(\emptyset) \to \mathit{FeatExp}(\mathbb{F})$, are specified as follows:

$$\boldsymbol{\alpha}^{\text{join}}(\psi) = \begin{cases} \mathit{true} & \text{if } \exists k \in \mathbb{K}. k \models \psi \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$\boldsymbol{\gamma}^{\text{join}}(\mathit{true}) = \mathit{true}, \qquad \boldsymbol{\gamma}^{\text{join}}(\mathit{false}) = \bigvee_{k \in 2^{\mathbb{F}} \setminus \mathbb{K}} k$$

**Theorem 1.** *$\langle \mathit{FeatExp}(\mathbb{F})_{/\equiv}, \models\rangle \underset{\boldsymbol{\alpha}^{\text{join}}}{\overset{\boldsymbol{\gamma}^{\text{join}}}{\leftrightarrows}} \langle \mathit{FeatExp}(\emptyset), \models\rangle$ is a Galois connection.*

*Proof.* Let $\psi \in \mathit{FeatExp}(\mathbb{F})$ and $\varphi' \in \mathit{FeatExp}(\emptyset)$.

$$\begin{aligned}
&\boldsymbol{\alpha}^{\text{join}}(\psi) \models \varphi' \\
&\iff (\exists k \in \mathbb{K}. k \models \psi \wedge \mathit{true} \models \varphi')\vee \\
&\qquad (\forall k \in \mathbb{K}. k \not\models \psi \wedge \mathit{false} \models \varphi') \quad (\text{by def. of } \boldsymbol{\alpha}^{\text{join}}) \\
&\iff (\exists k \in \mathbb{K}. k \models \psi \wedge \varphi' = \mathit{true})\vee \\
&\qquad (\forall k \in \mathbb{K}. k \not\models \psi \wedge (\varphi' = \mathit{true} \vee \varphi' = \mathit{false})) \quad (\text{by } \models) \\
&\iff \big((\exists k \in \mathbb{K}. k \models \psi \wedge \varphi' = \mathit{true})\vee \\
&\qquad (\forall k \in \mathbb{K}. k \not\models \psi \wedge \varphi' = \mathit{true})\big)\vee \\
&\qquad (\forall k \in \mathbb{K}. k \not\models \psi \wedge \varphi' = \mathit{false}) \quad (\text{by def. of } \vee) \\
&\iff (\psi \models \mathit{true} \wedge \varphi' = \mathit{true})\vee \\
&\qquad (\psi \models \bigvee_{k \in 2^{\mathbb{F}} \setminus \mathbb{K}} k \wedge \varphi' = \mathit{false}) \quad (\text{by def. of } \models) \\
&\iff \psi \models \boldsymbol{\gamma}^{\text{join}}(\varphi') \quad (\text{by def. of } \boldsymbol{\gamma}^{\text{join}})
\end{aligned}$$

□

*Ignoring features.* The abstraction $\boldsymbol{\alpha}_A^{\text{fignore}}$ ignores a single feature $A \in \mathbb{F}$ that is not directly relevant for the current analysis. We confound the control flow paths that only differ with regard to $A$, and keep the precision with respect to control flow paths that do not depend on $A$. Thus, $\boldsymbol{\alpha}_A^{\text{fignore}}$ merges any configurations that only differ with respect to $A$, and are identical with regard to remaining features, $\mathbb{F}\setminus\{A\}$.

Given a feature expression $\psi$, we write $\boldsymbol{\alpha}_A^{\text{fignore}}(\psi)$ for a weaker formula (such that $\psi \models \boldsymbol{\alpha}_A^{\text{fignore}}(\psi)$) obtained by eliminating the feature $A$ from $\psi$ in the following way. First, we convert $\psi$ into NNF (negation normal form), which contains only $\neg, \wedge, \vee$ connectives and $\neg$ appears only in literals. We write $l_A$ for the literals $A$ or $\neg A$. Then, we write $\boldsymbol{\alpha}_A^{\text{fignore}}(\psi) = \psi[l_A \mapsto \mathit{true}]$ to denote the formula $\psi$ where any literal of $A$, $l_A$, is replaced with $\mathit{true}$. Note that all formulae $k \in \mathbb{K}$ are already in NNF.

The abstract sets of features and valid configurations are: $\boldsymbol{\alpha}_A^{\text{fignore}}(\mathbb{F}) = \mathbb{F}\setminus\{A\}$, and $\boldsymbol{\alpha}_A^{\text{fignore}}(\mathbb{K}) = \{k[l_A \mapsto \mathit{true}] \mid k \in \mathbb{K}\}$. The abstraction and concretization functions between $\mathit{FeatExp}(\mathbb{F})$ and $\mathit{FeatExp}(\boldsymbol{\alpha}_A^{\text{fignore}}(\mathbb{F}))$ are defined as:

$$\begin{aligned}
\boldsymbol{\alpha}_A^{\text{fignore}}(\psi) &= \psi[l_A \mapsto \mathit{true}] \\
\boldsymbol{\gamma}_A^{\text{fignore}}(\varphi') &= (\varphi' \wedge A) \vee (\varphi' \wedge \neg A)
\end{aligned}$$

where $\psi$ and $\varphi'$ are in NNF from.

**Theorem 2.** $\langle \mathit{FeatExp}(\mathbb{F})_{/\equiv}, \models\rangle \underset{\boldsymbol{\alpha}_A^{\text{fignore}}}{\overset{\boldsymbol{\gamma}_A^{\text{fignore}}}{\leftrightarrows}}$ $\langle \mathit{FeatExp}(\mathbb{F}\setminus\{A\})_{/\equiv}, \models\rangle$ *is a Galois connection.*

[2] Alternatively, we could work with the set-theoretic definition of propositional formulae and consider an isomorphic complete lattice of sets of configurations $([\![\mathit{FeatExp}(\mathbb{F})]\!], \subseteq, \cup, \cap, 2^{\mathbb{F}}, \emptyset)$. Here, $[\![\mathit{FeatExp}(\mathbb{F})]\!] = 2^{2^{\mathbb{F}}}$, and an element of $[\![\mathit{FeatExp}(\mathbb{F})]\!]$ is a subset of $\mathit{Eval}(\mathbb{F}) = 2^{\mathbb{F}}$, which corresponds to a propositional formula over $\mathbb{F}$ which is satisfied by those evaluations.

*Proof.* Let $\psi \in FeatExp(\mathbb{F})$, and $\varphi' \in FeatExp(\mathbb{F}\backslash\{A\})$ be in NNF.

$$\begin{aligned} &\boldsymbol{\alpha}_A^{\text{fignore}}(\psi) \models \varphi' \\ &\iff \psi[l_A \mapsto true] \models \varphi' && \text{(by def. of } \boldsymbol{\alpha}_A^{\text{fignore}}) \\ &\iff \psi \models (\varphi' \wedge A) \vee (\varphi' \wedge \neg A) && \text{(by Lemma 1)} \\ &\iff \psi \models \boldsymbol{\gamma}_A^{\text{fignore}}(\varphi') && \text{(by def. of } \boldsymbol{\gamma}_A^{\text{fignore}}) \end{aligned}$$

□

**Lemma 1.** *Let $\psi$ and $\varphi'$ be in NNF. $\psi[l_A \mapsto true] \models \varphi'$ iff $\psi \models (\varphi' \wedge A) \vee (\varphi' \wedge \neg A)$.*

*Proof.* By induction on the structure of $\psi$.

*Sequential Composition.* The composition of two Galois connections gives also a Galois connection [18]. Let $\langle FeatExp(\mathbb{F})_{/\equiv}, \models\rangle \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \langle FeatExp(\alpha_1(\mathbb{F}))_{/\equiv}, \models\rangle$ and $\langle FeatExp(\alpha_1(\mathbb{F}))_{/\equiv}, \models\rangle \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \langle FeatExp(\alpha_2(\alpha_1(\mathbb{F})))_{/\equiv}, \models\rangle$ be two Galois connections. Then their composition is defined as:

$$\alpha_2 \circ \alpha_1(\psi) = \alpha_2(\alpha_1(\psi))$$
$$\gamma_1 \circ \gamma_2(\psi) = \gamma_1(\gamma_2(\psi))$$

We also have $\alpha_2 \circ \alpha_1(\mathbb{F}) = \alpha_2(\alpha_1(\mathbb{F}))$ and $\alpha_2 \circ \alpha_1(\mathbb{K}) = \alpha_2(\alpha_1(\mathbb{K}))$.

*Syntactic sugar.* We can define an operation which ignores a *set* of features: $\boldsymbol{\alpha}_{\{A_1,\ldots,A_m\}}^{\text{fignore}} = \boldsymbol{\alpha}_{A_1}^{\text{fignore}} \circ \ldots \circ \boldsymbol{\alpha}_{A_m}^{\text{fignore}}$ and $\boldsymbol{\gamma}_{\{A_1,\ldots,A_m\}}^{\text{fignore}} = \boldsymbol{\gamma}_{A_m}^{\text{fignore}} \circ \ldots \circ \boldsymbol{\gamma}_{A_1}^{\text{fignore}}$.

In the following, we will simply write $(\alpha, \gamma)$ for any $\langle FeatExp(\mathbb{F})_{/\equiv}, \models\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle FeatExp(\alpha(\mathbb{F}))_{/\equiv}, \models\rangle$, which is constructed using the operators presented in this section.

### 4.2 Abstracting FTSs

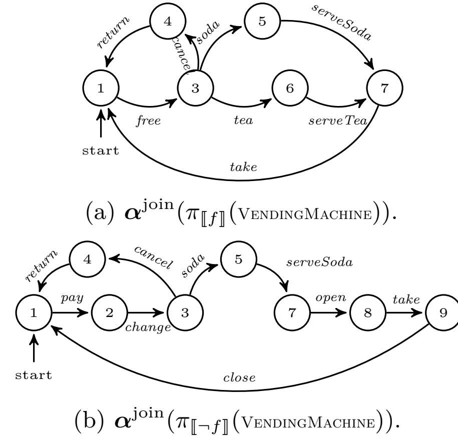
Given Galois connections defined on the level of feature expressions, available features, and valid configurations, we now induce a notion of abstraction between FTSs.

**Definition 4.** Let $\mathcal{F} = (S, Act, trans, I, AP, L, \mathbb{F}, \mathbb{K}, \delta)$ be an FTS, $[\chi]\phi$ be an fLTL formula, and $(\alpha, \gamma)$ be a Galois connection.

- We define $\alpha(\mathcal{F}) = (S, Act, trans, I, AP, L, \alpha(\mathbb{F}), \alpha(\mathbb{K}), \alpha(\delta))$, where $\alpha(\delta) : trans \to FeatExp(\alpha(\mathbb{F}))$ is defined as: $\alpha(\delta)(t) = \alpha(\delta(t))$.
- We define $\alpha([\chi]\phi) = [\alpha(\chi)]\phi$.

*Example 3.* Consider the FTS $\mathcal{F} =$ VendingMachine in Fig. 2 with $\mathbb{K} = \{\{v, s\}, \{v, s, t, f\}, \{v, s, c\}, \{v, s, c, f\}\}$. We have showed $\boldsymbol{\alpha}_{\{v,f\}}^{\text{fignore}}(\mathcal{F})$ and $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F})$ in Fig. 4. In Fig. 5 are also shown $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![f]\!]}(\mathcal{F}))$ and $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg f]\!]}(\mathcal{F}))$.

For $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![f]\!]}(\mathcal{F}))$ in Fig. 5a, note that $\mathbb{K} \cap [\![f]\!] = \{\{v, s, t, f\}, \{v, s, c, f\}\}$. So, transitions annotated with $v \wedge \neg f$ are not present in $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![f]\!]}(\mathcal{F}))$.




(a) $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![f]\!]}(\text{VendingMachine}))$.

(b) $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg f]\!]}(\text{VendingMachine}))$.

Fig. 5: Various abstractions of VendingMachine.

For $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![f]\!]}(\mathcal{F}))$ in Fig. 5b, note that $\mathbb{K} \cap [\![\neg f]\!] = \{\{v, s\}, \{v, s, c\}\}$, and so transitions annotated with the features $t$ and $f$ (`Tea` and `FreeDrinks`) are not present in $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg f]\!]}(\mathcal{F}))$. Also note that in the case of $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F})$, $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg f]\!]}(\mathcal{F}))$ and $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![f]\!]}(\mathcal{F}))$ we obtain ordinary transition systems, since all transitions are labeled with the feature expression *true*. □

### 4.3 Property Preservation

We now show that abstracted FTSs have some interesting preservation properties. In particular, we show an LTL formula satisfied by the abstracted FTS is also satisfied by the concrete FTS. First, we prove a helping lemma, which states that for any valid variant $k$ from the concrete FTS that can execute a behaviour guarded by feature expressions $\psi_0$, $\psi_1$, ..., there exists a variant $k'$ in the abstracted FTS that can execute the same behaviour.

**Lemma 2.** *Let $\chi, \psi_0, \psi_1, \ldots \in FeatExp(\mathbb{F})$, $\mathbb{K}$ be a set of configurations over $\mathbb{F}$, and $(\alpha, \gamma)$ be a Galois connection. Let $k \in \mathbb{K} \cap [\![\chi]\!]$, such that $k \models \psi_i$ for all $i \geq 0$. Then there exists $k' \in \alpha(\mathbb{K}) \cap [\![\alpha(\chi)]\!]$, such that $k' \models \alpha(\psi_i)$ for all $i \geq 0$.*

*Proof.* By induction on the structure of $\alpha$.

Case $\boldsymbol{\alpha}^{\text{join}}$: By assumption, we have that $\mathbb{K} \neq \emptyset$, thus $\boldsymbol{\alpha}^{\text{join}}(\mathbb{K}) = \{true\}$. By def. of $\boldsymbol{\alpha}^{\text{join}}$, we have that $\boldsymbol{\alpha}^{\text{join}}(\chi) = \boldsymbol{\alpha}^{\text{join}}(\psi_0) = \ldots = true$, since there exists $k \in \mathbb{K}$ such that $k \models \chi, k \models \psi_0, \ldots$. Since $true \in \boldsymbol{\alpha}^{\text{join}}(\mathbb{K})$ and $true \models true$, we obtain the conclusion.

Case $\boldsymbol{\alpha}_A^{\text{fignore}}$: By assumption, $k' = k[l_A \mapsto true] \in \boldsymbol{\alpha}_A^{\text{fignore}}(\mathbb{K})$. For any $\psi \in FeatExp(\mathbb{F})$, it holds: if $k \models \psi$ then $k[l_A \mapsto true] \models \psi[l_A \mapsto true]$. Thus, we have that $k' \in \boldsymbol{\alpha}_A^{\text{fignore}}(\mathbb{K}) \cap [\![\chi[l_A \mapsto true]]\!]$, and $k' \models \psi_i[l_A \mapsto true]$ for all $i \geq 0$.

Case $\alpha_2 \circ \alpha_1$: Follows directly by IH.

□

By using Lemma 2, we can prove the following result.

**Theorem 3 (Soundness).** *Let $(\alpha, \gamma)$ be a Galois connection.* $\alpha(\mathcal{F}) \models [\alpha(\chi)]\phi \implies \mathcal{F} \models [\chi]\phi$.

*Proof.* We proceed by contraposition. Assume $\mathcal{F} \not\models [\chi]\phi$. Then, there exist a configuration $k \in \mathbb{K} \cap [\![\chi]\!]$ and an execution $\rho = s_0\lambda_1 s_1 \lambda_2 \ldots \in [\![\pi_k(\mathcal{F})]\!]_{TS}$ such that $\rho \not\models \phi$, i.e. $\rho \models \neg\phi$. This means that for all transitions in $\rho$, $t_i = s_i \xrightarrow{\lambda_{i+1}} s_{i+1}$ for $i = 0, 1, \ldots$, we have that $k \models \delta(t_i)$ for all $i \geq 0$. By Lemma 2, we have that there exists $k' \in \alpha(\mathbb{K}) \cap [\![\alpha(\chi)]\!]$, such that $k' \models \alpha(\delta(t_i))$ for all $i \geq 0$. Hence, the execution $\rho$ is realizable for $\alpha(\mathcal{F})$, i.e. $\rho \in [\![\pi_{k'}(\alpha(\mathcal{F}))]\!]_{TS}$ and $\rho \models \neg\phi$. It follows that $\alpha(\mathcal{F}) \not\models [\alpha(\chi)]\phi$. □

It follows from Definition 3 that a family-based model checking problem can be reduced to a number of smaller problems by partitioning the set of variants:

**Proposition 1.** *Let the subsets $\mathbb{K}_1, \mathbb{K}_2, \ldots, \mathbb{K}_n$ form a* partition *of the set $\mathbb{K}$. Then: $\mathcal{F} \models [\chi]\phi$ iff $\pi_{\mathbb{K}_i}(\mathcal{F}) \models [\chi]\phi$ for all $i = 1, \ldots, n$.*

Using Proposition 1 and Theorem 3, we obtain the following result.

**Corollary 1.** *Let $\mathbb{K}_1, \mathbb{K}_2, \ldots, \mathbb{K}_n$ form a* partition *of $\mathbb{K}$, and $(\alpha_1,\gamma_1), \ldots, (\alpha_n,\gamma_n)$ be Galois connections. If $\alpha_1(\pi_{\mathbb{K}_1}(\mathcal{F})) \models [\alpha_1(\chi)]\phi, \ldots, \alpha_n(\pi_{\mathbb{K}_n}(\mathcal{F})) \models [\alpha_n(\chi)]\phi$, Then $\mathcal{F} \models [\chi]\phi$.*

The above results show that, *if* we are successfully able to verify an *abstracted* property for an *abstracted* FTS, then the verification also holds for the unabstracted (concrete) FTS. Note that verifying the abstracted FTS can be a lot (even exponentially) faster. If a counter-example is found in the abstracted FTS, then it may be spurious (introduced due to the abstraction) for some variants and genuine for the others. This can be established by checking which products can execute the found counter-example. By doing so, we are able to identify the set of products that violate a given property, and to report a counter-example of violation for each of them.

*Example 4.* Recall the formula $\phi = \texttt{G}(\texttt{selected} \implies \texttt{F}\,\texttt{open})$ from Example 2, and $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![f]\!]}(\textsc{VendingMachine}))$ and $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg f]\!]}(\textsc{VendingMachine}))$ shown in Fig. 5. First, we can successfully verify that $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg f]\!]}(\textsc{VendingMachine})) \models \phi$, which implies that all valid variants that do not contain the feature $f$ satisfy the property $\phi$. On the other hand, we have $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![f]\!]}(\textsc{VendingMachine})) \not\models \phi$. This means that the feature $f$ (or a more complex feature interaction in which $f$ is always enabled) is responsible for the property $\phi$ not to hold in the VendingMachine. In this way, the problem of verifying whether the FTS VendingMachine satisfies $\phi$ is reduced to verifying whether two ordinary TSs, $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg f]\!]}(\textsc{VendingMach.}))$ and $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![f]\!]}(\textsc{VendingMach.}))$, satisfy $\phi$. □

In general, we aim to partition the set of variants $\mathbb{K}$ into subsets $\mathbb{K}_1, \ldots, \mathbb{K}_n$ such that all variants of any subset either satisfy the property at hand or do not satisfy it.

## 5 High-Level Modelling Languages

It is very difficult to use FTSs to directly model very large systems. Therefore, it is necessary to have a high-level modelling language, which can be used directly by engineers for modelling large variational systems. *f*Promela is designed for describing variational systems; whereas TVL for describing the sets of features and valid configurations. We present *f*Promela and TVL and show their FTS semantics.

### 5.1 Syntax

*f*Promela is obtained from Promela [30] by adding *feature variables*, $\mathbb{F}$, and *guarded-by-features statements*. Promela is a non-deterministic modelling language designed for describing systems composed of concurrent processes that communicate asynchronously. A Promela model, $P$, consists of a finite set of processes to be executed concurrently. The basic statements of processes are given by:

$$\begin{aligned} stm ::= {} & \texttt{skip} \mid \texttt{x := } expr \mid c?x \mid c!expr \mid stm_1 \texttt{;} stm_2 \mid \\ & \texttt{if} :: g_1 \Rightarrow stm_1 \cdots :: g_n \Rightarrow stm_n :: \texttt{else} \Rightarrow stm\, \texttt{fi} \mid \\ & \texttt{do} :: g_1 \Rightarrow stm_1 \cdots :: g_n \Rightarrow stm_n\, \texttt{od} \end{aligned}$$

where $x$ is a variable, $c$ is a channel, and $g_i$ are conditions over variables and contents of channels. The "`if`" is a non-deterministic choice between the statements $stm_i$ for which the guard $g_i$ evaluates to *true* for the current evaluation of the variables. If none of the guards $g_1, \ldots, g_n$ are *true* in the current state, then the "`else`" statement $stm$ is chosen. Similarly, the "`do`" represents an iterative execution of the non-deterministic choice among the statements $stm_i$ for which the guard $g_i$ holds in the current state. Statements are preceded by a declarative part, where variables and channels are declared.

The feature variables, $\mathbb{F}$, used in an *f*Promela (variational) model have to be declared as fields of the special type *features*. The new guarded-by-features statement introduced in *f*Promela is of the form:

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

where $\psi_1, \ldots, \psi_n$ are feature expressions defined over $\mathbb{F}$. The "`gd`" is a non-deterministic statement similar to "`if`", except that only features can be used as conditions (guards). Actually, this is the only place where features may be used. Hence, "`gd`" in *f*Promela plays the same role as "`#ifdef`" in C/CPP SPLs [33].

*TVL* [10] is a textual modelling language for describing the set of all valid configurations, $\mathbb{K}$, for an *f*Promela model along with all available features, $\mathbb{F}$. A feature model is organized as a tree, whose nodes denote features and edges represent parent-child relationship between nodes. The `root` keyword denotes the root of the tree, and the `group` keyword, followed by a decomposition type "`allOf`", "`someOf`", or "`oneOf`", declares

the children of a node. The meaning is that if the parent feature is part of a variant, then "all", "some", or "exactly one" respectively, of its non-optional children have to be part of that variant. The optional features are preceded by the `opt` keyword. Various Boolean constraints on the presence of features can be specified as well.

*Example 5.* Fig. 6 shows simple *f*Promela and TVL models. After declaring feature variables in the *f*Promela model in Fig. 6a, the process `foo` is defined. The first `gd` statement specifies that `i++` is available for variants that contain the feature `A`, and `skip` for variants with ¬`A`. The second `gd` statement is similar, except that the guard is the feature `B`. The TVL model in Fig. 6b (single line comments start with "//") specifies four valid configurations: {`Main`}, {`Main`, `A`}, {`Main`, `B`}, {`Main`, `A`, `B`}. If we use the SNIP tool to check the assertion, `i` ≥ `0`, in this example, we will obtain that it is satisfied by all (four) valid variants. However, the assertion `i` > `0` will fail for the variant ¬`A`∧¬`B` (i.e. `Main`), where both features `A` and `B` are disabled. If we include the constraint in comments in line 5 of Fig. 6b that excludes the variant: ¬`A` ∧ ¬`B`, then the assertion `i` > `0` will also hold for the given family, which now contains three valid variants. □

### 5.2 Semantics

We now show only the most relevant details of the *f*Promela semantics. For the precise account of the Promela semantics the reader is referred to [30]. Each *f*Promela model defines a so-called *featured program graph* (FPG), which formalizes the control flow of the model. The FPG represents a program graph [3] (or "finite state automaton" in [30]) in which transitions are explicitly linked with feature expressions. The vertices of the graph are control locations (represented by line numbers in the model) and its transition relation defines the control flow of the model. Each transition has *condition* under which it can be executed, an *effect* which specifies its effect on the set of variables, and a *feature expression* which indicates in which variants this transition is enabled. Thus, transitions are annotated with condition/effect/feature expression. The "`gd`" statement specifies the control flow and the feature expression part of transitions.

Let $V$ be the set of variables, and $\mathbb{F}$ be the set of features in an *f*Promela model. Let $Cond(V)$ denote the set of Boolean conditions over $V$, and $Assgn(V)$ denote all assignments over $V$. $Eval(V)$ is the set of all evaluations of $V$ that assign concrete values to variables in $V$. For $v \in Eval(V)$ and $g \in Cond(V)$, we write $v \models g$ if the evaluation $v$ makes $g$ *true*, and $apply(a, v)$ is the evaluation obtained after applying the assignment $a \in Assgn(V)$ to $v$.

**Definition 5.** A FPG over $V$ and $\mathbb{F}$ is defined as a tuple $(Loc, tr, Loc_0, init, \mathbb{K}, fe)$, where $Loc$ is a set of control locations, $Loc_0 \subseteq Loc$ is a set of initial locations, $tr \subseteq Loc \times Cond(V) \times Assgn(V) \times Loc$ is the transition relation, $init \in Cond(V)$ is the initial condition characterising the variables in the initial state, $\mathbb{K}$ is a set of configurations, and $fe : tr \to FeatExp(\mathbb{F})$ annotates transitions with feature expressions.

The *semantics* of an FPG is an *FTS* obtained from "*unfolding*" the graph (see [3, Sect. 2] for details). The unfolded FTS is:

$$(Loc \times Eval(V), \{\epsilon\}, trans, I, Cond(V), L, \mathbb{F}, \mathbb{K}, \delta)$$

where the states are pairs of the form $(l, v)$ for $l \in Loc, v \in Eval(V)$; action names are ignored ($\epsilon$ is an empty, dummy, action name); $I = \{(l, v) \mid l \in Loc_0, v \models init\}$; $L((l, v)) = \{g \in Cond(V) \mid v \models g\}$; and transitions are defined as:

$$\frac{(l, g, a, l') \in tr \qquad v \models g}{((l, v), \epsilon, (l', apply(a, v))) \in trans}$$

Given $t \in trans$, let $t' \in tr$ be the corresponding transition of the FPG. Then $\delta(t) = true$ if $fe(t')$ is undefined; and $\delta(t) = fe(t')$ otherwise. Hence, the semantics of *f*Promela follows the semantics of Promela, just adding feature expression from the FPG to transitions.

*Example 6.* In Fig. 7 are shown the FPG and FTS for the family model defined in Fig. 6. As shown in the FPG in Fig. 7a, a "`gd`" statement becomes a state with one outgoing transition per option, so that the last state of all options leads back to a common state. For instance, see transitions from ⑮ (line 5) and ⑯ (line 6). The unfolded FTS is shown in Fig. 7b, where each state contains the info about the control location (line number) and the current value of the variable `i`. □

## 6 Syntactic Transformations

We present the syntactic transformations of *f*Promela and TVL models introduced by projection and variability abstractions. Let $P$ represent an *f*Promela model, for which the sets of features $\mathbb{F}$ and valid configurations $\mathbb{K}$ are given as a TVL model $T$. We denote with $[\![P]\!]_T$ the FTS obtained for this program, as shown in Section 5.

Let $\psi'$ be a feature expression, such that $[\![\psi']\!] \subseteq \mathbb{K}$. The projection $\pi_{[\![\psi']\!]}([\![P]\!]_T)$ is obtained by adding the constraint $\psi'$ in the corresponding TVL model $T$, which we denote as $T + \psi'$. Thus, $\pi_{[\![\psi']\!]}([\![P]\!]_T) = [\![P]\!]_{T+\psi'}$.

Let $(\alpha, \gamma)$ be a Galois connection obtained from our calculus in Section 4. The abstract $\alpha(P)$ and $\alpha(T)$ are obtained by defining a translation recursively over the structure of $\alpha$. The function $\alpha$ copies all non-compound basic statements of *f*Promela, and recursively calls itself for all sub-statements of compound statements other than "`gd`". For example, $\alpha(\texttt{skip}) = \texttt{skip}$ and $\alpha(stm_1;stm_2) = \alpha(stm_1);\alpha(stm_2)$. We discuss the rewrites for "`gd`" below.

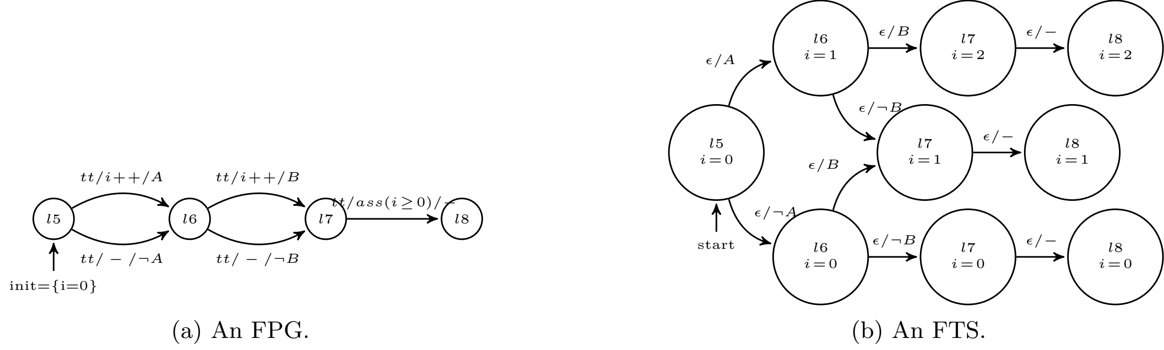
```
0 typedef features {
1   bool A; bool B; }
2 features f;
3 active proctype foo() {
4   int i := 0;
5   gd :: f.A ⇒ i++ :: else ⇒ skip dg;
6   gd :: f.B ⇒ i++ :: else ⇒ skip dg;
7   assert(i ≥ 0);
8 }
```

(a) An *f*Promela model.

```
0 root Main {
1   group allOf {
2     opt A,
3     opt B
4   }
5   // A || B;
6 }
```

(b) A TVL model.

Fig. 6: Simple *f*Promela and TVL models




(a) An FPG. (b) An FTS.

Fig. 7: The semantics of the family system defined in Fig. 6. Note that "*lx*" refers to the line number $x$ from the program in Fig. 6a, and *tt* is short for *true*.

For $\boldsymbol{\alpha}^{\text{join}}$, we obtain a Promela (single variant) model $\boldsymbol{\alpha}^{\text{join}}(P)$ where all "gd"-s are appropriately resolved and all features are removed. Thus, $\boldsymbol{\alpha}^{\text{join}}(T)$ is empty. The transformation is

$$\begin{aligned}
&\boldsymbol{\alpha}^{\text{join}}\big(\texttt{gd} :: \psi_1 \Rightarrow stm_1 \ldots \\
&\qquad :: \psi_n \Rightarrow stm_n :: \texttt{else} \Rightarrow stm' \, \texttt{dg}\big) = \\
&\quad \texttt{if} :: \boldsymbol{\alpha}^{\text{join}}(\psi_1) \Rightarrow \boldsymbol{\alpha}^{\text{join}}(stm_1) \ldots \\
&\quad :: \boldsymbol{\alpha}^{\text{join}}(\psi_n) \Rightarrow \boldsymbol{\alpha}^{\text{join}}(stm_n) \\
&\quad :: \boldsymbol{\alpha}^{\text{join}}(\neg(\psi_1 \vee \ldots \vee \psi_n)) \Rightarrow \boldsymbol{\alpha}^{\text{join}}(stm') \, \texttt{fi}
\end{aligned}$$

For $\boldsymbol{\alpha}_A^{\text{fignore}}$, the transformation for "gd" is:

$$\begin{aligned}
&\boldsymbol{\alpha}_A^{\text{fignore}}\big(\texttt{gd} :: \psi_1 \Rightarrow stm_1 \ldots \\
&\qquad :: \psi_n \Rightarrow stm_n :: \texttt{else} \Rightarrow stm' \, \texttt{dg}\big) = \\
&\quad \texttt{gd} :: \boldsymbol{\alpha}_A^{\text{fignore}}(\psi_1) \Rightarrow \boldsymbol{\alpha}_A^{\text{fignore}}(stm_1) \ldots \\
&\quad :: \boldsymbol{\alpha}_A^{\text{fignore}}(\psi_n) \Rightarrow \boldsymbol{\alpha}_A^{\text{fignore}}(stm_n) \\
&\quad :: \boldsymbol{\alpha}_A^{\text{fignore}}(\neg(\psi_1 \vee \ldots \vee \psi_n)) \Rightarrow \boldsymbol{\alpha}_A^{\text{fignore}}(stm') \, \texttt{dg}
\end{aligned}$$

The new $\boldsymbol{\alpha}_A^{\text{fignore}}(T)$ is obtained by removing the feature $A$ from $T$, when $\mathbb{F} \backslash \{A\} \neq \emptyset$. Otherwise, if $\mathbb{F} \backslash \{A\} = \emptyset$, then $\boldsymbol{\alpha}_A^{\text{fignore}}(P)$ is a Promela model where the keyword "gd" is replaced with "if", and $\boldsymbol{\alpha}_A^{\text{fignore}}(T)$ is empty.

For $\alpha_2 \circ \alpha_1$, we have $\alpha_2 \circ \alpha_1(\texttt{gd} :: \psi_1 \Rightarrow stm_1 \ldots \texttt{dg}) = \alpha_2(\alpha_1(\texttt{gd} :: \psi_1 \Rightarrow stm_1 \ldots \texttt{dg}))$. Similarly, we transform the TVL model $T$, i.e. $\alpha_2 \circ \alpha_1(T) = \alpha_2(\alpha_1(T))$.

Since feature expressions are treated in the same way by FTS and *f*Promela, we can show that the relation between *f*Promela and abstract *f*Promela models coincides with that between FTSs and abstracted FTSs.

**Theorem 4.** *Let $P$ and $T$ be fPromela and TVL models, and $(\alpha, \gamma)$ be a Galois connection. We have: $\alpha([\![P]\!]_T) \equiv [\![\alpha(P)]\!]_{\alpha(T)}$, where $\equiv$ means that both FTSs are semantically equivalent.*

*Proof.* By induction on the structure of $P$ and *stm*. The only interesting case is for the "gd" statement, since in all other cases we have an identity translation.

For the "gd" statement, we can see that all abstractions $\alpha$ are applied on feature expressions, which can be introduced in FTSs only through "gd"-s. Thus, it is the same whether the abstraction is applied directly on FTSs after the FTS is built by following the operational semantics of "gd", or the abstraction is first applied on "gd" statements and after that the corresponding FTS is built. □

## 7 Evaluation

We now evaluate our verification technique based on variability abstractions on several case studies. The evaluation aims to address the following objectives:

- **O1:** To show how variability abstractions can turn a previously infeasible analysis of model families into a feasible one;
- **O2:** That instead of verifying properties using a family-based model checker (e.g., $\overline{\text{SNIP}}$), we can use variability abstraction to obtain an abstract family-of-models

(with a low number of variants) that can then be efficiently model checked using a single-system model checker (e.g., SPIN);

**O3:** To demonstrate that we can use variability abstractions to speed-up verification of some properties using only a family-based model checker (e.g., $\overline{\text{SNIP}}$), by ignoring features that are not relevant for the given properties.

To achieve the above objectives, we use the soundness of variability abstractions (Theorem 3). That is, *if* we are able to verify properties on the abstract model family, we may safely conclude that they also hold on the original (unabstract) model family.

### 7.1 Experimental setup

All of our abstractions are applied using our *f*PROMELA RECONFIGURATOR (source-*TO*-source) transformation tool[3] as described in Section 6. We investigate improvements in *performance* (TIME) and *memory consumption* (SPACE) on two benchmarks: MINEPUMP and ELEVATOR family-models [11,12], that come with the installation of $\overline{\text{SNIP}}$. We do a case study on both benchmarks by verifying a range of properties using $\overline{\text{SNIP}}$ and SPIN model checkers. We show how various variability abstractions may be tailored for analysis of properties of the MINEPUMP and the ELEVATOR.

All experiments were executed on a 64-bit Mac OS X 10.10 machine, Intel®Core$^{TM}$ i7 CPU running at 2.3 GHz with 8 GB memory. The performance numbers reported (TIME) constitute the median runtime of five independent executions. For each experiment, we report: TIME which is the time to model check in seconds; and SPACE which is the number of explored states plus the number of re-explored states (this is equivalent to the number of transitions fired). In the case of $\overline{\text{SNIP}}$, the verification time includes the times to parse the *f*PROMELA model, to build the initial FTS, and to run the verification procedure. In the case of SPIN, we measure the times to generate a process analyser (pan) and to execute it. We do not count the time for compiling pan, as it is due to a design decision in SPIN rather than its verification algorithm. The same measurement technique was used in the experiments in [11,12].

### 7.2 Warming-up example: from infeasible to feasible analysis

Combinatorically, the number of variant models grows exponentially with the number of features, $|\mathbb{F}|$, which means that there is an inherent exponential blow-up in the analysis time for the brute-force strategy, $\mathcal{O}(2^{|\mathbb{F}|})$. Consequently, for families with high variability, analysis

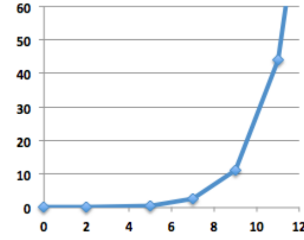



Fig. 8: The performance of "brute-force" family-based model checking with SPIN as a function of the number of features. The x-axis represents the number of features, and the y-axis represents the verification time in seconds.

quickly becomes infeasible. They take too long time to analyze.

As an experiment, we have tested the limits of family-based model checking with $\overline{\text{SNIP}}$ and "brute force" (single-system) model checking with SPIN, where *all* variants of a given model family are verified one by one. We have gradually added variability to the family-model in Fig. 6. This was done by adding unconstrained optional features and by sequentially composing gd statements guarded by all existing features. For example, the *f*PROMELA process *foo* with three features *A*, *B*, and *C* is:

```
int i := 0;
gd :: f.A ⇒ i++ :: else ⇒ skip dg;
gd :: f.B ⇒ i++ :: else ⇒ skip dg;
gd :: f.C ⇒ i++ :: else ⇒ skip dg;
assert(i ≥ 0);
```

Already for $|\mathbb{F}| = 11$ (for which $|\mathbb{K}| = 2^{11} = 2,048$ variants) $\overline{\text{SNIP}}$ crashes with an out-of-memory error, whereas analysis time to check the assertion using "brute force" with SPIN becomes almost a minute. For $|\mathbb{F}| = 25$, analysis time ascends to almost a year. Figure 8 depicts this phenomenon. It shows the accumulated time (in seconds) of using SPIN to "brute-force" verify the assertion "$i \geq 0$" on *all* individual variant models of the family-model in Fig 6, sequentially, for increasing number of features, $|\mathbb{F}|$. On the other hand, if we apply the variability abstraction, $\boldsymbol{\alpha}^{\text{join}}$ (confounding all configurations), prior to analysis, we are able to verify the same assertion by only one call to SPIN on the *abstracted* model in 0.03 seconds for $|\mathbb{F}| = 11$ and in 0.04 seconds for $|\mathbb{F}| = 25$, effectively eliminating the exponential blow up (cf. Objective **O1**).

### 7.3 Case studies: devising abstractions for properties

We perform case studies on two benchmarks: MINEPUMP and ELEVATOR, by showing how various variability abstractions can be tailored for their analysis.

#### 7.3.1 MINEPUMP

***Characterization.*** The MINEPUMP system was introduced in the CONIC project [34]. Based on the original

[3] The *f*PROMELA RECONFIGURATOR tool is available from: https://models-team.github.io/p3-tool/.

| $\vert\mathbb{F}\vert$ | *unabstracted* $\vert\mathbb{K}\vert$ | Time | Space | *abstracted* $\vert\alpha(\mathbb{K})\vert$ | Time | Space | *improvement* Time | Space |
|---|---|---|---|---|---|---|---|---|
| 4 | 16 | 0.98 s | 67 k | 1 | 0.03 s | 18 k | 33 × | 3.8 × |
| 7 | 128 | 2.96 s | 251 k | 1 | 0.04 s | 34 k | 74 × | 7.4 × |
| 9 | 512 | 6.05 s | 523 k | 1 | 0.05 s | 57 k | 121 × | 9.2 × |
| 11 | 2,048 | 58.55 s | 4,585 k | 1 | 0.07 s | 114 k | 836 × | 40.3 × |
| 12 | 4,096 | — ⋆crash⋆ — | | 1 | 0.09 s | 171 k | *infeasible* → *feasible* | |

Fig. 9: Verifying deadlock absence in MinePump for increasing levels of variability (without vs. with maximal abstraction, $\alpha = \boldsymbol{\alpha}^{\text{join}}$, confounding all configurations).

system, an *f*Promela model was created in [12] as part of the $\overline{\text{SNIP}}$ project. The *f*Promela 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.

***Verification.*** We start by verifying the deadlock freedom property. Fig. 9 compares the effect (in terms of both Time and Space) of analyzing the original (unabstracted) MinePump vs. analyzing it after it has been variability abstracted using $\boldsymbol{\alpha}^{\text{join}}$. *Unabstracted* means running $\overline{\text{SNIP}}$ on MinePump; whereas *abstracted* means running SPIN on $\boldsymbol{\alpha}^{\text{join}}$(MinePump). *Improvement* is the relative comparison of *unabstracted* vs. *abstracted*. The rows of Fig. 9 represent different versions of MinePump, with increasing levels of variability. The "real" version has $|\mathbb{K}| = 128$ variants. For the $|\mathbb{K}| = 16$ version, we applied a *projection* to keep the four features `Start`, `Stop`, `MethaneAlarm`, and `High` (eliminating features `MethaneQuery`, `Low`, and `Normal`). For the $|\mathbb{K}| = 512$ version, we turned implementation alternatives (already present in the original MinePump, as comments) into variability choices in the form of two new independent features. Parts of the `controller` process exists *with* and *without* race conditions (the former in comments); we turned that into an optional feature, `RaceCond`. Similarly, the `watersensor` process exists in two versions: *standard* and *alternative* (the latter in comments); we turned that into an optional feature, `Standard`. For $|\mathbb{K}| = 2,048$ and $|\mathbb{K}| = 4,096$, we inflated variability by adding independent optional features and `gd` statements to the `methanesensor` process, preserving the overall behavior of the process (differing only with respect to the value of an otherwise uninteresting local variable, `i`).

Unsurprisingly, analysis Time and Space increase exponentially with the number of features, $\mathcal{O}(|\mathbb{F}|)$. However, the Time and Space it takes to verify the deadlock absence in the *abstracted* model do not increase significantly with the number of variants, when using the maximal abstraction, $\boldsymbol{\alpha}^{\text{join}}$. For $|\mathbb{K}| = 2,048$ variants, $\overline{\text{SNIP}}$ terminates after almost a minute (checking 4.6 million transitions) whereas calling SPIN on the *abstracted* system obtains the verification results after a mere 0.07 seconds visiting only 113,775 transitions. For $|\mathbb{K}| = 4,096$ variants, $\overline{\text{SNIP}}$ *crashes* after 88 seconds exploring 6.3 million transitions. SPIN, on the other hand, is capable of analyzing the *abstracted* system in 0.09 seconds exploring 170,670 transitions (cf. Objectives **O1** and **O2**).

| $\Phi$ | *property* |
|---|---|
| $\varphi_0$ | `(GF readCommand)` ∧ `(GF readAlarm)` ∧ `(GF readLevel)`<br>*Fairness: The system will infinitely often read messages of various types.* |
| $\varphi_1$ | *Absence of deadlock.* |
| $\varphi_2$ | `G (¬pumpOn ∨ stateRunning)`<br>*If the pump is switched on, then the controller state is "running".* |
| $\varphi_3$ | $\varphi_0$ ⇒ `(¬GF (¬pumpOn ∧¬methane ∧ highWater))`<br>*Assuming fairness ($\varphi_0$), the pump is never indefinitely off when the water level is high and there is no methane.* |
| $\varphi_4$ | `G ((¬pumpOn ∧ lowWater ∧ F highWater) ⇒ (¬pumpOn U highWater))`<br>*When the pump is off and the water level is low then the the pump will be switched off until the water level is high again.* |
| $\varphi_5$ | `¬(GF pumpOn)`<br>*The pump cannot be switched on infinitely often* |
| $\varphi_6$ | $\varphi_0$ ⇒ `¬FG (pumpOn ∧ methane)`<br>*Assuming fairness ($\varphi_0$), the system cannot be in a state where the pump is on indefinitely in the presence of methane.* |

Fig. 10: Properties for the MinePump (taken from [12]).

Now we continue by verification of some interesting properties of MinePump. First, we consider four universal properties, $\varphi_1$ to $\varphi_4$ (taken from [12], see Fig. 10), that are intended to be satisfied by all variants. Property $\varphi_0$ is an auxiliary *fairness assumption* used in $\varphi_3$; and then later in $\varphi_6$. By applying the $\boldsymbol{\alpha}^{\text{join}}$ abstraction on the system, we can verify those properties efficiently by only *one* call to SPIN on the *abstracted* family-model, $\boldsymbol{\alpha}^{\text{join}}$(MinePump) which has only one con-

| *prop--erty* | *unabstracted* $\lvert\mathbb{K}\rvert$ | Time | Space | *abstracted* $\lvert\alpha(\mathbb{K})\rvert$ | Time | Space | *improvement* Time | Space |
|---|---|---|---|---|---|---|---|---|
| $\varphi_1$ | 128 | 2.96 s | 251 k | 1 | 0.04 s | 34 k | 74 × | 7.4 × |
| $\varphi_2$ | 128 | 4.28 s | 326 k | 1 | 0.05 s | 34 k | 86 × | 9.6 × |
| $\varphi_3$ | 128 | 6.37 s | 441 k | 1 | 0.09 s | 161 k | 71 × | 2.7 × |
| $\varphi_4$ | 128 | 5.98 s | 420 k | 1 | 0.05 s | 57 k | 120 × | 7.3 × |
| $\varphi_5$ | 128 | 3.20 s | 207 k | 3 | 0.11 s | 12 k | 29 × | 16.6 × |
| $\varphi_6$ | 128 | 4.54 s | 309 k | 4 | 0.16 s | 42 k | 28 × | 7.3 × |

Fig. 11: Verification of MinePump properties using tailored abstractions.

figuration, $|\boldsymbol{\alpha}^{\text{join}}(\mathbb{K}_{\text{MinePump}})| = 1$. The first four rows of Fig. 11 organizes the results of maximally abstracting the MinePump prior to verfication of properties, $\varphi_1$ to $\varphi_4$. Consistent with our expectations and previous results (cf. Fig. 9), maximal abstraction translates to massive improvements in both Time and Space on a family-model with many variants (here, $|\mathbb{K}| = 128$). In fact, model checking is between 71 and 120 times faster (cf. Objective **O2**).

We now consider non-universal properties which are *preserved* by some variants and *violated* by others: $\varphi_5$ and $\varphi_6$ (see Fig. 10). Property $\varphi_5$ (concerning the pump being switched on), is violated by all variants, 32 in total, for which `Start` $\wedge$ `High` is satisfied (since these two features are required for the pump to be switched on in the first place). Given sufficient knowledge of the system and the property, we can easily tailor an abstraction for analyzing the system more effectively: First, we calculate three projections of the MinePump family-model: $\pi_{\texttt{Start}\wedge\texttt{High}}$ (corresponding to the above 32 configurations), $\pi_{\neg\texttt{Start}}$ (64 configurations), and $\pi_{\neg\texttt{High}}$ (64 configurations). Second, we apply $\boldsymbol{\alpha}^{\text{join}}$ on all three projections. Third and finally, we invoke SPIN three times to verify $\varphi_5$ on each of them. For the first abstracted projection, $\boldsymbol{\alpha}^{\text{join}}(\pi_{\texttt{Start}\wedge\texttt{High}}(\text{MinePump}))$, SPIN correctly identifies an *"abstract"* counter-example violating the property, that is *shared* by all violating variants. For the remaining abstracted projections, SPIN reports that $\varphi_5$ is *satisfied*.

We now turn to $\varphi_6$ (again involving the pump, this time in the presence of methane). Since the features `Start` and `High` are required for the pump to be on (as in $\varphi_5$), this violation now occurs when the `MethaneAlarm` feature is disabled in a variant. Property $\varphi_6$ is thus violated whenever `Start`$\wedge$`High`$\wedge\neg$`MethaneAlarm` is *satisfied* (corresponding to 16 variants, in total). As before, we may concoct our abstraction: We first calculate (this time four) projections of the system: $\pi_{\texttt{Start}\wedge\texttt{High}\wedge\neg\texttt{MethaneAlarm}}$, $\pi_{\neg\texttt{Start}}$, $\pi_{\neg\texttt{High}}$, and $\pi_{\texttt{MethaneAlarm}}$. We then apply $\boldsymbol{\alpha}^{\text{join}}$ on all projections. Finally, we invoke SPIN (four times) to verify $\phi_6$. We obtain that $\phi_6$ is violated by the first abstracted projection along with an *"abstract"* counter-example, *shared* by all violating products. Further, SPIN finds that $\phi_6$ is satisfied for the remaining abstractions.

Overall, we can see that our approach is significantly faster (cf. Objective **O2**). The second-last row of Fig. 11 shows that analysis time drops from 3.20 seconds when $\varphi_5$ verified with $\overline{\text{SNIP}}$, to 0.11 seconds when running SPIN "brute-force" on our *three* abstracted projections. The last row shows the results of a similar development for the property, $\varphi_6$. It takes 4.54 seconds using $\overline{\text{SNIP}}$, but may be verified by *four* "brute-force" invocations of SPIN in only 0.16 seconds. Verification of both properties constitute an almost 30 times speed up (using considerably less memory).

We can also use $\boldsymbol{\alpha}^{\text{fignore}}$ abstraction to speed up the family-based model checker itself (cf. Objective **O3**). For property $\varphi_5$, we call $\overline{\text{SNIP}}$ on $\boldsymbol{\alpha}^{\text{fignore}}_{\mathbb{F}\setminus\{\texttt{Start},\texttt{High}\}}(\text{MinePump})$, and we obtain the same counter-examples as in the unabstracted case for the variants in `Start` $\wedge$ `High`. However, the verification time is reduced from 3.20 to 0.97 seconds, and the number of examined transitions is reduced from $207,377$ to $54,376$. Similarly, for property $\varphi_6$, we call $\overline{\text{SNIP}}$ on $\boldsymbol{\alpha}^{\text{fignore}}_{\mathbb{F}\setminus\{\texttt{Start},\texttt{High},\texttt{MethaneAlarm}\}}(\text{MinePump})$ obtaining that $\varphi_6$ is violated by products described with: `Start`$\wedge$`High`$\wedge\neg$`MethaneAlarm`. The verification time is now reduced from 4.54 to 1.34 seconds, and the number of examined transitions is reduced from $308,812$ to $79,450$.

#### 7.3.2 Elevator

***Characterization.*** The Elevator was designed by Plath and Ryan [40]. The corresponding $f$Promela model was created in [11,17]. The $f$Promela Elevator family contains about 300 LOC and 8 (non-mandatory) independent optional features: `Empty`, `Exec`, `OpenIfIdle`, `Overload`, `Park`, `QuickClose`, `Shuttle`, and `TTFull`, thus yielding $2^8 = 256$ variants. Its FTS has 58,945,690 states. It consists of 3 communicating processes: a `controller`, and two `persons`. It serves a number of floors (which is four in our case) such that the priority is given to the nearest floor in current direction movement.

***Verification.*** We consider four properties [17], $\varphi_1$ to $\varphi_4$, shown in Fig. 12. The property $\varphi_1$ is satisfied by all variants. Thus, by applying the $\boldsymbol{\alpha}^{\text{join}}$ abstraction on Elevator, we can verify this property by only *one* call to SPIN. The property $\varphi_2$ (resp., $\varphi_3$) is violated by prod-

| $\Phi$ | property |
|---|---|
| $\varphi_1$ | `GF progress` $\Rightarrow$ $(\neg$`FG dopen`$)$ <br> *The door should never remain open indefinitely.* |
| $\varphi_2$ | $\neg$`F ((cb0` $\vee$ `cb1` $\vee$ `cb2` $\vee$ `cb3)` $\wedge$ <br> $\neg$ `(p0in` $\wedge$ `p1in)` $\wedge$ `dclosed)` <br> *It is impossible that cabin buttons are pressed and nobody is inside.* |
| $\varphi_3$ | $\neg$`F (p0in` $\wedge$ `p1in` $\wedge$ `dclosed))` <br> *There cannot be two persons in the elevator at the same time.* |
| $\varphi_4$ | `GF (progress` $\vee$ `waiting)` $\Rightarrow$ $(\neg$`FG dopen`$)$ <br> *The same as* $\varphi_1$ *but accounting for the waiting time.* |

Fig. 12: Properties for the Elevator.

ucts for which $\neg$`Empty` (resp., $\neg$`Overload`) is satisfied. We can verify both by two calls to SPIN. For $\varphi_2$, we verify $\boldsymbol{\alpha}^{\text{join}}(\pi_{\neg\texttt{Empty}}(\text{Elevator}))$ and $\boldsymbol{\alpha}^{\text{join}}(\pi_{\texttt{Empty}}(\text{Elevator}))$. For the first abstracted projection, SPIN reports an 'abstract' counter-example, whereas the second abstracted projection satisfies $\varphi_2$. In a similar way, we can verify the property $\varphi_3$ by constructing projections $\pi_{\neg\texttt{Overload}}$ and $\pi_{\texttt{Overload}}$. The property $\varphi_4$ (concerning that the door never remains open indefinitely) is violated by products that satisfy: `Park`$\vee$`OpenIfIdle`$\vee\neg$`QuickClose`. If these features are present, then the lift may keep its door indefinitely open since persons might keep pushing buttons indefinitely or stop. We use SPIN to verify satisfaction of $\varphi_4$ against four models obtained by applying $\boldsymbol{\alpha}^{\text{join}}$ on the following projections of Elevator: $\pi_{\neg\texttt{Park}\wedge\neg\texttt{OpenIfIdle}\wedge\texttt{QuickClose}}$, $\pi_{\texttt{Park}}$, $\pi_{\texttt{OpenIfIdle}}$, and $\pi_{\neg\texttt{QuickClose}}$. We obtain that $\varphi_4$ is satisfied by the first abstracted projection, and is violated by the remaining three abstracted projections.

We can see in Fig. 13 that abstractions achieve impressive speed-ups in both Time and Space. The verifications are performed between 114 and 1570 times faster (cf. Objective **O2**). For the property $\varphi_2$, the analysis with $\overline{\text{SNIP}}$ ran more than 135 seconds until it eventually produced an out-of-memory error, whereas with 2 calls to SPIN we completed the verification in 0.35 seconds (cf. Objective **O1**).

We are also able to use $\boldsymbol{\alpha}^{\text{fignore}}$ abstraction to speed up verification with $\overline{\text{SNIP}}$. For property $\varphi_3$, we call $\overline{\text{SNIP}}$ on $\boldsymbol{\alpha}^{\text{fignore}}_{\mathbb{F}\setminus\{\texttt{Overload}\}}(\text{Elevator})$, which contains only 2 products, and we obtain that the `Overload` product satisfies $\varphi_3$ whereas $\neg$`Overload` violates $\varphi_3$. The verification time compared to the concrete model Elevator is reduced from 197.23 to 6.14 seconds, and the number of examined transitions is reduced from 33,204,216 to 896,392 (cf. Objective **O3**).

### *7.4 Discussion*

In conclusion, by exploiting the knowledge of a family-model and property, we may carefully devise variability abstractions that are able to verify non-trivial properties in only a few calls to SPIN.

Of course, much of the performance improvement is due to the highly-optimized industrial-strength SPIN tool compared to the $\overline{\text{SNIP}}$ research prototype. SPIN contains many optimisation algorithms, which are result of more than three decades research on advanced computer aided verification. For example, partial order reduction, data-flow analysis and statement merging are not implemented in $\overline{\text{SNIP}}$ yet. Previous work attributes a factor of two advantage for "brute force" approach with SPIN over $\overline{\text{SNIP}}$ [12]. However, for models with more variability (larger values of $|\mathbb{F}|$), a constant factor will be dwarfed by the inherent exponential blow up.

## 8 A Note on Automatic Verification

As we have discussed in Section 7.4, a user of our approach needs to have a good knowledge of a variational model and property in order to manually devise variability abstractions that will enable efficient verification. We now give an overview of an algorithm, which aims to automate our verification approach. It is based on an abstraction refinement procedure (ARP), which iteratively refines an abstract variational model until either a genuine counter-example is found or the property satisfaction is shown. The ARP for checking $\mathcal{F} \models [\chi]\phi$, where $\mathcal{F}{=}(S, Act, trans, I, AP, L, \mathbb{F}, \mathbb{K}, \delta)$, is as follows.

1 Let $\alpha = \boldsymbol{\alpha}^{\text{join}}$ be the initial abstraction used to build $\alpha(\mathcal{F})$ and $\alpha(\chi)$. Check $\alpha(\mathcal{F}) \models [\alpha(\chi)]\phi$?
2 If the property is satisfied, then return that $\phi$ is satisfied for all products in $\mathbb{K} \cap [\![\chi]\!]$.
3 Otherwise, if a spurious counter-example is found, find the feature expression associated with its transitions in $\mathcal{F}$: $\psi_1 \wedge \ldots \wedge \psi_n$. Since the counter-example is spurious, the formula $\psi = (\bigvee_{k\in\mathbb{K}} k) \wedge \chi \wedge \psi_1 \wedge \ldots \wedge \psi_n$ is unsatisfiable. First, we find the minimal unsatisfiable core $\psi^c$ of $\psi$, which contains a subset of conjuncts in $\psi$, such that $\psi^c$ is still unsatisfiable and if we drop any single conjunct in $\psi^c$ then the result becomes satisfiable. We group conjuncts in $\psi^c$ in two groups $X$ and $Y$ such that $\psi^c = X \wedge Y = false$. Then the feature expression $\psi'$ is determined by means of Craig interpolation [38]. Namely, the interpolant $\psi'$ is such that: 1) $X \implies \psi'$, 2) $\psi' \wedge Y = false$, 3) $\psi'$ refers only to common variables of $X$ and $Y$. Intuitively, we can think of the interpolant $\psi'$ as a way of filtering out irrelevant information from $X$. In particular, $\psi'$ summarizes and translates why $X$ is inconsistent with $Y$ in their shared language. We generate $\mathcal{F}_1{=}\pi_{[\![\psi']\!]}(\mathcal{F})$ and $\mathcal{F}_2{=}\pi_{[\![\neg\psi']\!]}(\mathcal{F})$, and go to Step 1 to check $\mathcal{F}_1 \models$

| *prop-erty* | *unabstracted* $\lvert\mathbb{K}\rvert$ | TIME | SPACE | *abstracted* $\lvert\alpha(\mathbb{K})\rvert$ | TIME | SPACE | *improvement* TIME | SPACE |
|---|---|---|---|---|---|---|---|---|
| $\varphi_1$ | 256 | 753.62 s | 110 M | 1 | 0.48 s | 1.25 M | 1570 × | 88 × |
| $\varphi_2$ | 256 | — ⋆crash⋆ — | | 2 | 0.35 s | 0.54 M | *infeasible* → *feasible* | |
| $\varphi_3$ | 256 | 197.23 s | 33.2 M | 2 | 0.39 s | 0.64 M | 505 × | 52 × |
| $\varphi_4$ | 256 | 63.94 s | 9.13 M | 4 | 0.56 s | 0.65 M | 114 × | 14 × |

Fig. 13: Verification of ELEVATOR properties using tailored abstractions.

$[\chi]\phi$ for products in $\mathbb{K}_1 = [\![\psi']\!] \cap \mathbb{K}$ and $\mathcal{F}_2 \models [\chi]\phi$ for products in $\mathbb{K}_2 = [\![\neg\psi']\!] \cap \mathbb{K}$.

4 Otherwise, if a genuine counter-example is found, find the feature expression associated with its transitions in $\mathcal{F}$: $\psi = \psi_1 \wedge \ldots \wedge \psi_n$. Report that the property is violated for products in $\mathbb{K} \cap [\![\chi]\!] \cap [\![\psi]\!]$. We generate $\mathcal{F}' = \pi_{[\![\neg\psi]\!]}(\mathcal{F})$, and go to Step 1 to check $\mathcal{F}' \models [\chi]\phi$ for products in $\mathbb{K}' = [\![\neg\psi]\!] \cap \mathbb{K}$.

*Example 7.* Recall the FTS $\mathcal{F}$ = VENDINGMACHINE in Fig. 2 with $\mathbb{K} = \{\{v,s\},\{v,s,t,f\},\{v,s,c\},\{v,s,c,f\}\}$. Let us check the formula $\phi = \mathtt{G}\,(\mathtt{selected} \implies \mathtt{F}\,\mathtt{open})$ using the ARP. We first check $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F}) \models \phi$? Assume that the spurious counter-example ① $\xrightarrow{pay}$ ② $\xrightarrow{change}$ ③ $\xrightarrow{tea}$ ⑥ $\xrightarrow{serveTea}$ ⑦ $\xrightarrow{take}$ ① ... is reported. The associated feature expression is: $(v \wedge \neg f) \wedge v \wedge t \wedge f$. The minimal unsatisfiable core is: $(v \wedge \neg f) \wedge f$, and the found interpolant is $\neg f$. In this way, we have found that the feature $f$ is responsible for the spuriousness of the given counter-example. Thus, in the next iteration we check $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg f]\!]}(\mathcal{F})) \models \phi$ and $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![f]\!]}(\mathcal{F})) \models \phi$, as explained in Example 4. □

## 9 Related Work

In the last decade, researchers have introduced family-based (lifted) analyses and verification techniques for program families (software product lines). They work at the family level and thus do not explicitly check all variants, one by one (see [44] for survey). Family-based techniques are able to pinpoint errors directly in the family, as opposed to reporting errors in individual variants. We divide our discussion of related work into several categories: family-based model checking on FTSs; abstractions for family-based model checking on FTSs; other family-based model checking approaches; other family-based abstractions; and family-based static analysis.

***Family-based model checking on FTSs.*** Classen et al. have proposed FTSs in [14,12] as the foundation for behavioural specification and verification of variational systems. In [11], they show how the theory on (explicit) family-based model checking algorithms for verifying FTSs against fLTL properties is implemented in the $\overline{\text{SNIP}}$ model checker. In [13], Classen et al. present symbolic family-based model checking algorithms for verifying FTSs against fCTL properties. The algorithms are implemented as an extension of the NuSMV model checker [8]. Variational models are specified using the fSMV modelling language introduced by Plath and Ryan [40], which is feature-oriented extension of the input language of NuSMV. However, in contrast to $\overline{\text{SNIP}}$, this model checker reports a counter-example only for the first violating product found. The FMC tool [42] is used for family-based model checking of FTSs with a $\mu$-calculus variant, where FTSs are automatically transformed into a process-algebraic model that can directly serve as input for the FMC. In order to make all these approaches based on FTSs more scalable abstractions need to be applied.

***Abstractions for family-based model checking on FTSs.*** *Simulation-based abstraction* for family-based model checking was introduced in [16]. The concrete FTS is related with its abstract version by defining a *simulation* relation on the level of states (as opposed to Galois connections here). Several abstract (and thus smaller) models are induced by studying quotients of concrete FTSs under such a simulation relation. Any behaviour of the concrete FTS can be reproduced in its abstraction, and therefore the abstraction preserves satisfiability of LTL formulae. Only states and transitions that can be simulated are reduced by this approach. However, this approach [16] results in small model reductions and only marginal efficiency gains of verifications times (the evaluation reports reductions of 8-9%). Since abstractions are applied directly on FTSs, the computation time for calculating abstracted FTSs takes about 10% of the overall verification time. This approach has been extended in [4], where the definition of branching feature bisimulation for FTSs is given. A minimization algorithm is also presented to compute, given an FTS, a minimal FTS that is a branching feature bisimilar.

Variability-aware abstraction procedures based on counterexample guided abstraction refinement (CEGAR) have been proposed in [17]. Abstractions on FTSs are introduced by using existential F-abstraction functions, and simulation relation is used to relate different abstraction levels. Three types of FTSs abstractions are considered: *state abstractions* that only merge states, *feature abstractions* that only modify the variability information, and *mixed abstractions* that combine the previous two types. Feature abstractions [17] are similar to ours since they also aim to reduce variability specific information in SPLs. However, there are many differences between them.

Different levels of precision of feature abstractions in [17] are defined by simply enriching (resp., reducing) the sets of variants for which transitions are enabled. In contrast, our variability abstractions are capable to change not only the feature expression labels of transitions but also the sets of available features and valid configurations. Moreover, the user can use those abstractions to express various verification scenarios for their families. While the abstractions in [17] are applied on FPGs, we apply our abstractions as preprocessor transformations directly on high-level modelling languages thus avoiding to generate any intermediate concrete semantic model in the memory.

***Other family-based model checking approaches.*** One of the earliest approaches for modelling behavioral variability is by using *modal transition systems* (MTSs) [26,43] or *variable I/O automata* [35]. In MTSs, transitions can be mandatory ("must") or optional ("may"), such that optional transitions are used to model variability. Beek et al. [43] have implemented a model checking tool, called VMC, for the specification and verification of behavioral variability in product lines modelled as MTSs with logical variability constraints, where the properties are expressed as v-ACTL formulae. Lauenroth et al. [35] have proposed an explicit CTL model checking algorithm for product lines specified with variable I/O automata.

Process-algebraic approaches have also been proposed for modelling and verifying the behaviour of product lines. PL-CCS [29] is an extension of Milner's process algebra CCS, which is enriched with a variant operator as a means to implement variability. Delta-CCS [36] is another delta-oriented extension of CCS, in which variability is achieved by decomposing the product line into a core process and a set of delta modules that encapsulate change directives based on term rewriting semantics. Model checking algorithms are also described in [29,36] for system families specified in both PL-CCS and Delta-CCS against properties specified in modal $\mu$-calculus.

Variability encoding [2] and configuration lifting [41] are based on generating a family *simulator* which simulates the behaviour of all variants in the family. Both approaches use a classical off-the-shelf model checker to verify the generated family simulator, so that an erroneous violating variant can be reconstructed/decoded from a found counter-example. A problem with these approaches is that they stop once a single counter-example is found. Thus, they cannot be used to find the variants that satisfy the given property. This also makes it difficult to find which features are responsible for the violations.

An approach for family-based software model checking using game semantics has been proposed in [22]. Specifically designed family-based model checking algorithms are employed for verifying safety of `#ifdef`-based programs containing undefined components (free identifiers), which are compactly represented using symbolic game semantics models [21].

***Other family-based abstractions.*** Thüm et. al. in [46] use *variability hiding* to facilitate the efficient deductive verification based on method contracts of two dependent product lines (implemented using feature modules). They propose three strategies to hide features of one product line for the other product line. The first two strategies, namely *false*- and *true-configuration*, verify only a subset of all possible configurations (where given features are disabled or enabled) of the reused product line, and thus they closely resemble to our projection operator. The third strategy, *hidden-configuration*, removes features by considering that the given features can be both disabled and enabled. In this way, this strategy is similar to our $\boldsymbol{\alpha}^{\mathrm{fignore}}_{A}$ operator, which generates an over-approximation of an FTS by assuming that the feature $A$ have both possible values.

***Family-based Static Analysis.*** As mentioned before, various lifted techniques have been proposed, which *lift* existing analysis and verification techniques to work on the level of families, rather than on the level of single programs/systems. This includes lifted syntax checking [33], lifted type checking [7,32], lifted data-flow analysis [6,5], lifted theorem proving [45], etc.

A formal methodology for systematic derivation of lifted data-flow (static) analyses for program families with `#ifdef`-s is proposed in [39]. The method uses the calculational approach to abstract interpretation of Cousot [18] in order to derive a directly operational lifted analysis. In [24,25], an expressive calculus of variability abstractions is also devised for deriving abstracted lifted data-flow analyses. Such variability abstractions enable deliberate trading of precision for speed in lifted data-flow analysis. Hence, they tame the exponential blow-up caused by the large number of features and variants in an program family. We show the effectiveness of that approach by evaluating two client data-flow analyses: reaching definitions and uninitialized variables, on three real-world Java program families. Here, we pursue this line of work by adapting variability abstractions to lifted model checking as opposed to data-flow analysis in [24]. Moreover, the abstractions in [24] are directed at reducing the configuration space $|\mathbb{K}|$ since the elements of the property domain are $|\mathbb{K}|$-sized tuples, whereas the abstractions defined here aim at reducing the space of feature expressions since the variability-sensitive information in FTSs, fLTL formulae, and $f$PROMELA models is encoded by using feature expressions.

## 10 Conclusion

We have proposed variability abstractions to derive abstract model checking for families of related systems. The abstractions are applied before semantic model generation directly on $f$PROMELA defined system-families as

source to source transformations. The evaluation confirms that interesting properties can be efficiently verified in this way by only a few calls to SPIN. As a future work, we plan to implement and experiment with the abstraction refinement procedure for automatic generation of suitable abstractions.

## References


1. Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer, 2013.
2. Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. Detection of feature interactions using feature-aware verification. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 372–375, 2011.
3. Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.
4. Tessa Belder, Maurice H. ter Beek, and Erik P. de Vink. Coherent branching feature bisimulation. In *Proceedings 6th Workshop on Formal Methods and Analysis in SPL Engineering, FMSPLE 2015*, volume 182 of *EPTCS*, pages 14–30, 2015.
5. Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. Spl$^{\text{lift}}$: statically analyzing software product lines in minutes instead of years. In *ACM SIGPLAN Conference on PLDI '13*, pages 355–364, 2013.
6. Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, Johnni Winther, and Paulo Borba. Intraprocedural dataflow analysis for software product lines. *Transactions on Aspect-Oriented Software Development*, 10:73–108, 2013.
7. Sheng Chen, Martin Erwig, and Eric Walkingshaw. An error-tolerant type system for variational lambda calculus. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12*, pages 29–40, 2012.
8. Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A new symbolic model checker. *STTT*, 2(4):410–425, 2000.
9. Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.
10. Andreas Classen, Quentin Boucher, and Patrick Heymans. A text-based approach to feature modelling: Syntax and semantics of TVL. *Sci. Comput. Program.*, 76(12):1130–1143, 2011.
11. Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. Model checking software product lines with SNIP. *STTT*, 14(5):589–612, 2012.
12. Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-François Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. *IEEE Trans. Software Eng.*, 39(8):1069–1089, 2013.
13. Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic model checking of software product lines. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011*, pages 321–330, 2011.
14. Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. 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*, pages 335–344, 2010.
15. Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.
16. Maxime Cordy, Andreas Classen, Gilles Perrouin, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. Simulation-based abstractions for software product-line model checking. In *Proceedings of the 34th International Conference on Software Engineering, ICSE 2012*, pages 672–682, 2012.
17. Maxime Cordy, Patrick Heymans, Axel Legay, Pierre-Yves Schobbens, Bruno Dawagne, and Martin Leucker. 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)*, pages 190–201, 2014.
18. Patrick Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.
19. Krzysztof Czarnecki and Michal Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *Generative Programming and Component Engineering, 4th Int. Conf., GPCE 2005*, volume 3676 of *LNCS*, pages 422–437, 2005.
20. Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, 1997.
21. Aleksandar S. Dimovski. Program verification using symbolic game semantics. *Theor. Comput. Sci.*, 560:364–379, 2014.
22. Aleksandar S. Dimovski. Symbolic game semantics for model checking program families. In *Model Checking Software - 23nd International Symposium, SPIN 2016, Proceedings*, volume 9641 of *LNCS*, pages 19–37. Springer, 2016.
23. Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wasowski. Family-based model checking without a family-based model checker. In *Model Checking Software - 22nd International Symposium, SPIN 2015, Proceedings*, volume 9232 of *LNCS*, pages 282–299. Springer, 2015.
24. Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Variability abstractions: Trading precision for speed in family-based analyses. In *29th European Conference on Object-Oriented Programming, ECOOP 2015*, volume 37 of *LIPIcs*, pages 247–270. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.
25. Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Variability abstractions: Trading precision for speed in family-based analyses (extended version). *CoRR*, abs/1503.04608, 2015.
26. Alessandro Fantechi and Stefania Gnesi. A behavioural model for product families. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference*

*and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007*, pages 521–524. ACM, 2007.
27. María-del-Mar Gallardo, Jesús Martínez, Pedro Merino, and Ernesto Pimentel. $\alpha$spin: A tool for abstract model checking. *STTT*, 5(2-3):165–184, 2004.
28. María-del-Mar Gallardo, Pedro Merino, and Ernesto Pimentel. Refinement of LTL formulas for abstract model checking. In *Static Analysis, 9th International Symposium, SAS 2002, Proceedings*, volume 2477 of *LNCS*, pages 395–410. Springer, 2002.
29. Alexander Gruler, Martin Leucker, and Kathrin D. Scheidemann. Modeling and model checking software product lines. In *Formal Methods for Open Object-Based Distributed Systems, 10th IFIP WG 6.1 International Conference, FMOODS 2008, Proceedings*, volume 5051 of *LNCS*, pages 113–131. Springer, 2008.
30. Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.
31. Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.
32. Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. Type checking annotation-based product lines. *ACM Trans. Softw. Eng. Methodol.*, 21(3):14, 2012.
33. Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 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*, pages 805–824, 2011.
34. Jeff Kramer, Jeff Magee, Morris Sloman, and Andrew Lister. Conic: An integrated approach to distributed computer control systems. *IEE Proc.*, 130(1):1–10, 1983.
35. Kim Lauenroth, Klaus Pohl, and Simon Toehning. Model checking of domain artifacts in product line engineering. In *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, 2009*, pages 269–280. IEEE Computer Society, 2009.
36. Malte Lochau, Stephan Mennicke, Hauke Baller, and Lars Ribbeck. Incremental model checking of delta-oriented software product lines. *J. Log. Algebr. Meth. Program.*, 85(1):245–267, 2016.
37. Claire Loiseaux, Susanne Graf, Joseph Sifakis, Ahmed Bouajjani, and Saddek Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995.
38. Kenneth L. McMillan. Applications of craig interpolants in model checking. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Proceedings*, volume 3440 of *LNCS*, pages 1–12. Springer, 2005.
39. Jan Midtgaard, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Systematic derivation of correct variability-aware program analyses. *Sci. Comput. Program.*, 105:145–170, 2015.
40. Malte Plath and Mark Ryan. Feature integration using a feature construct. *Sci. Comput. Program.*, 41(1):53–84, 2001.
41. Hendrik Post and Carsten Sinz. Configuration lifting: Verification meets software configuration. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, pages 347–350, 2008.
42. Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, and Franco Mazzanti. Using FMC for family-based analysis of software product lines. In *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015*, pages 432–439. ACM, 2015.
43. Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, and Franco Mazzanti. Modelling and analysing variability in product families: Model checking of modal transition systems with variability constraints. *J. Log. Algebr. Meth. Program.*, 85(2):287–315, 2016.
44. Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, 47(1):6, 2014.
45. Thomas Thüm, Ina Schaefer, Martin Hentschel, and Sven Apel. Family-based deductive verification of software product lines. In *Generative Programming and Component Engineering, GPCE'12*, pages 11–20. ACM, 2012.
46. Thomas Thüm, Tim Winkelmann, Reimar Schröter, Martin Hentschel, and Stefan Krüger. Variability hiding in contracts for dependent software product lines. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems, 2016*, pages 97–104. ACM, 2016.