Family-Based Model Checking without a Family-Based Model Checker [⋆]

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

IT University of Copenhagen, Denmark

Abstract. Many software systems are variational: they can be configured to meet diverse sets of requirements. Variability is found in both communication protocols and discrete controllers of embedded systems. In these areas, model checking is an important verification technique. For variational models (systems with variability), specialized *family-based* model checking algorithms allow efficient verification of multiple variants, simultaneously. These algorithms scale much better than “brute force” verification of individual systems, one-by-one. Nevertheless, they can deal with only very small variational models.

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 abstractions are first defined as Galois connections on semantic domains. We then show how to translate them into syntactic source-to-source transformations on variational models. This allows the use of SPIN with all its accumulated optimizations for efficient verification of variational models without any knowledge about variability. We demonstrate the practicality of this method on several examples using both the $\overline{\text{SNIP}}$ (family based) and SPIN (single system) model checkers.

1 Introduction

Variability is an increasingly frequent phenomenon in software systems. A growing number of projects follow the *Software Product Line* (SPL) methodology [8] for building a *family* of related systems. Implementations of such systems usually [1] contain statically configured options (variation points) governed by a *feature configuration*. A feature configuration determines a single *variant* (product) of the system family, which can be derived, built, tested, and deployed. The SPL methodology is particularly popular in the embedded systems domain, where development and production in lines is very common (e.g., cars, phones) [8].

Variability plays a significant role outside of the SPL methodology as well. Many communication protocols, components and system-level programs are

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

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 highly configurable systems is similar to static SPLs. Thus, systems with variability, i.e. system families, can be conceptually specified using *variational* models.

Since embedded systems, system-level software and communication protocols frequently are safety critical, they require rigorous validation of models, where model-checking is a primary validation technique. Performance of single-variant (single-system) model checking algorithms depends on the size of the model and the size of the specification property [2]. Classical model-checking research provides abstraction and reduction techniques to address the complexity stemming from both the model and the specification [4,13,15]. In most of these works, the generation of the abstract model is based on abstract interpretation theory [11]: the semantics of the concrete model is related with the semantics of its abstract version by using Galois connections. Provided 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.

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 (aka, 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 algorithms have been proposed [6,7]. However, efficiency of these algorithms *still* depend on the size of the configuration space (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 third issue—the size of the configuration space.

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. Then, we define source-to-source transformations on the source level of input models, which enable an effective computation of abstract models syntactically from high-level modelling languages. This makes it easier to implement than using the semantic-based abstractions defined for (featured) transition systems [7]. We avoid the need for intermediate storage in memory of the semantics of the concrete variational model. It also opens up a possibility of verifying properties of variational models, so of multiple model variants simultanously, without using a dedicated family-based model checker such as $\overline{\text{SNIP}}$ [6] (overlined purely to avoid confusion with SPIN). 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 (e.g., SPIN [16]).

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 syntactic 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 the application of abstractions in a preprocessing step, without any modifications to the model checking tools.
- **Family-based model-checking w/o a family-based model checker:** A method for Family-based model-checking using an off-the-shelf model-checker. This method relies on partitioning and abstracting the variational models until the point when they contain no variability. The default highly-optimized implementation of SPIN can 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. 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.

2 Background: Variational Models of Behavior

A common way of introducing variability into modeling languages is superimposing multiple variants in a single model [12]. Following this, Classen et al. present *f*PROMELA [6], an extension of PROMELA with a static configuration-time branching capable of *enabling/disabling* model code in variants. They generalize the semantic model of PROMELA (transition systems) accordingly, including static guard conditions over features on transitions, creating *featured transition systems* (FTS). The guards determine in which variants the transitions appear. The set of legal configurations is encoded in a separate so-called *feature model* [17]. They have also proposed model-checking algorithms for verification of FTSs against LTL properties and implemented them in the $\overline{\text{SNIP}}$ tool[1].

Featured Transition Systems (FTS). Let $\mathbb{F} = \{A_1, \ldots, A_n\}$ be a finite set of Boolean variables representing the features available in a variational model. A

[1] https://projects.info.unamur.be/fts/

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.) In software engineering the set of valid configurations is typically described by a feature model [17], but we disregard syntactic representations of the set $\mathbb{K}$ in this paper.

Example 1. Throughout this paper, we use the beverage VendingMachine example [7]. It contains the following features: `VendingMachine`, denoted v, for purchasing a drink which represents a mandatory root feature enabled in all variants; `Tea` (t), for serving tea; `Soda` (s), for serving soda; `CancelPurchase` (c), for canceling a purchase after a coin is entered; and `FreeDrinks` (f) for offering free drinks. Hence, $\mathbb{F} = \{v, t, s, c, f\}$. In this example, we assume that only configurations in the set $\mathbb{K} = \{\{v, s\}, \{v, s, t, c, f\}, \{v, t, c\}, \{v, t, c, f\}\}$ are valid. The valid configuration $\{v, s\}$ can be expressed as the formula $v \wedge s \wedge \neg t \wedge \neg c \wedge \neg f$.

The behaviour of individual variants is given as transition systems.

Definition 1. *A transition system is a tuple* $\mathcal{T} = (S, \mathit{Act}, \mathit{trans}, I, AP, L)$*, where* S *is a set of states,* Act *is a set of actions,* $\mathit{trans} \subseteq S \times \mathit{Act} \times S$ *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* $(s_1, \lambda, s_2) \in \mathit{trans}$.

An *execution* of a transition system $\mathcal{T}$ is a nonempty, potentially infinite sequence $\rho = s_0 \lambda_1 s_1 \lambda_2 \ldots$ 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 $[\![\mathcal{T}]\!]_{\mathrm{TS}}$, is the set of all executions of $\mathcal{T}$.

Let *FeatExp*($\mathbb{F}$), denote the set of all Boolean constraints over $\mathbb{F}$ generated using the grammar: $\psi ::= \mathit{true} \mid A \in \mathbb{F} \mid \neg\psi \mid \psi_1 \wedge \psi_2$. For a condition $\psi \in \mathit{FeatExp}(\mathbb{F})$ we write $[\![\psi]\!]$ meaning the set of valid variants that satisfy ψ, i.e. $k \in [\![\psi]\!]$ iff $k \models \psi$ and $k \in \mathbb{K}$, where $\models$ denotes the standard satisfaction of propositional logic. Feature transition systems are basically transition systems 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 feature transition system (FTS) 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. $[\![\mathcal{F}]\!]_{\mathrm{FTS}} = \bigcup_{k \in \mathbb{K}} [\![\pi_k(\mathcal{F})]\!]_{\mathrm{TS}}$. The *size* of an FTS [7] is defined as: $|\mathcal{F}| = |S| + |\mathit{trans}| + |\mathit{expr}| +$

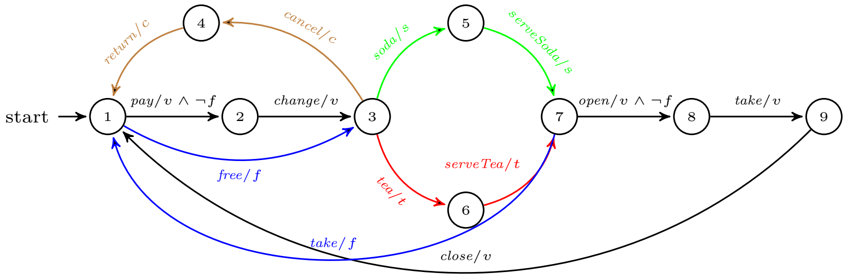


Fig. 1: FTS for the vending machine.

$|\mathbb{K}|$, where $|expr|$ is the size of all feature expressions bounded by $O(2^{|\mathbb{F}|} \cdot |trans|)$. In these terms, our abstractions aim to reduce the $|expr|$ and $|\mathbb{K}|$ components of the size $|\mathcal{F}|$.

Example 2. Figure 1 presents an FTS describing the behavior of the VENDING-MACHINE. Each transition is labeled first by an action, and then by the feature expression following a slash. For readability, the transitions included by the same feature have the same color. The transition $\textcircled{3} \xrightarrow{soda/s} \textcircled{5}$ is enabled by feature s. A basic variant of this machine that only serves soda is defined by the configuration $\{v, s\}$. It accepts payment, returns change, serves a soda, opens the access compartment, so that the customer can take the soda, and closes it again.

The fLTL Logics and Properties. An LTL formula is defined as: $\phi ::= true \mid a \in AP \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid X\phi \mid \phi_1\, U \phi_2$, with the following standard satisfaction semantics defined over an execution $\rho = s_0\lambda_1 s_1 \lambda_2 \ldots$ (we write $\rho_i = s_i \lambda_i s_{i+1} \ldots$ for the i-th suffix of ρ):

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

A TS $\mathcal{T}$ satisfies a formula ϕ, written $\mathcal{T} \models \phi$, iff $\forall \rho \in [\![\mathcal{T}]\!]_{\text{TS}} :\ \rho \models \phi$. Other temporal operators can be derived as usual: $\mathtt{F}\,\phi = true\,\mathtt{U}\,\phi$ (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 ϕ is an LTL formula and $\chi \in FeatExp(\mathbb{F})$ is a feature expression. 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 ϕ iff $\mathcal{F} \models [true]\phi$.*

Example 3. Consider the FTS $\mathcal{F}$ in Fig. 1. Suppose that states ⑤ and ⑥ are labeled *selected*, and the state ⑧ is labeled *open*. Consider an example property ϕ that after each time a beverage has been selected, the machine will eventually open the compartment to allow the customer to access his drink: $\mathtt{G}\,(\mathit{selected} \implies \mathtt{F}\,\mathit{open})$. The basic vending machine satisfies this property: $\pi_{\{v,s\}}(\mathcal{F}) \models \phi$, while the entire variational model does not: $\mathcal{F} \not\models \phi$. For example, if the feature f is enabled, the state ⑧ is unreachable. At the same time, we have that $\mathcal{F} \models [\neg f]\phi$.

3 Variability Abstractions

We shall now introduce abstractions decreasing the sizes of FTSs, in particular the number of features and the configuration space. We show how these abstractions preserve fLTL properties allowing to speed-up the algorithms for model checking. The proofs of all theorems and lemmas in this section can be found in App. A.

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. 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 ψ obtained by quotienting by the semantic equivalence $\equiv$. (Alternatively, we could work with the set-theoretic definition of propositional formulae and an isomorphic complete lattice of sets of configurations, without changing the theory in any way.)

Join. This abstraction confounds the control-flow of all configurations 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 ψ in the FTS is replaced with *true* if ψ 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.

A pair of *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:

$$\boldsymbol{\alpha}^{\text{join}}(\psi) = \begin{cases} \mathit{true} & \text{if } \exists k \in \mathbb{K}. k \models \psi \\ \mathit{false} & \text{otherwise} \end{cases} \qquad \begin{array}{l} \boldsymbol{\gamma}^{\text{join}}(\mathit{true}) = \mathit{true} \\ \boldsymbol{\gamma}^{\text{join}}(\mathit{false}) = \bigvee_{k \in 2^{\mathbb{F}} \setminus \mathbb{K}} k \end{array}$$

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*[2].

[2] $\langle L, \leq_L \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle M, \leq_M \rangle$ is a *Galois connection* between complete lattices L and M iff α and γ are total functions that satisfy: $\alpha(l) \leq_M m \iff l \leq_L \gamma(m)$ for all $l \in L, m \in M$.

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 we keep the precision with respect to control flow paths that do not depend on A.

To apply this abstraction, we first need to convert the given feature expression ψ 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 $\neg A$. We write $\psi[l_A \mapsto true]$ to denote the formula ψ where l_A is replaced with $true$.

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

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

where ψ and φ' are in NNF from.

Theorem 2. *$\langle FeatExp(\mathbb{F})_{/\equiv}, \models\rangle \xleftarrow[\boldsymbol{\alpha}_A^{\text{fignore}}]{\boldsymbol{\gamma}_A^{\text{fignore}}} \langle FeatExp(\mathbb{F}\backslash\{A\})_{/\equiv}, \models\rangle$ is a Galois connection.*

Sequential Composition. The composition of two Galois connections is also a Galois connection [11]. Let $\langle FeatExp(\mathbb{F})_{/\equiv}, \sqsubseteq\rangle \xleftrightarrow[\alpha_1]{\gamma_1} \langle FeatExp(\alpha_1(\mathbb{F}))_{/\equiv}, \sqsubseteq\rangle$ and $\langle FeatExp(\alpha_1(\mathbb{F}))_{/\equiv}, \sqsubseteq\rangle \xleftrightarrow[\alpha_2]{\gamma_2} \langle FeatExp(\alpha_2(\alpha_1(\mathbb{F})))_{/\equiv}, \sqsubseteq\rangle$ be two Galois connections. Then $\langle FeatExp(\mathbb{F})_{/\equiv}, \sqsubseteq\rangle \xleftrightarrow[\alpha_2\circ\alpha_1]{\gamma_1\circ\gamma_2} \langle FeatExp(\alpha_2(\alpha_1(\mathbb{F})))_{/\equiv}, \sqsubseteq\rangle$ 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 $\boldsymbol{\gamma}_{\{A_1,\ldots,A_m\}}^{\text{fignore}} = \boldsymbol{\gamma}_{A_m}^{\text{fignore}} \circ \ldots \circ \boldsymbol{\gamma}_{A_1}^{\text{fignore}}$.

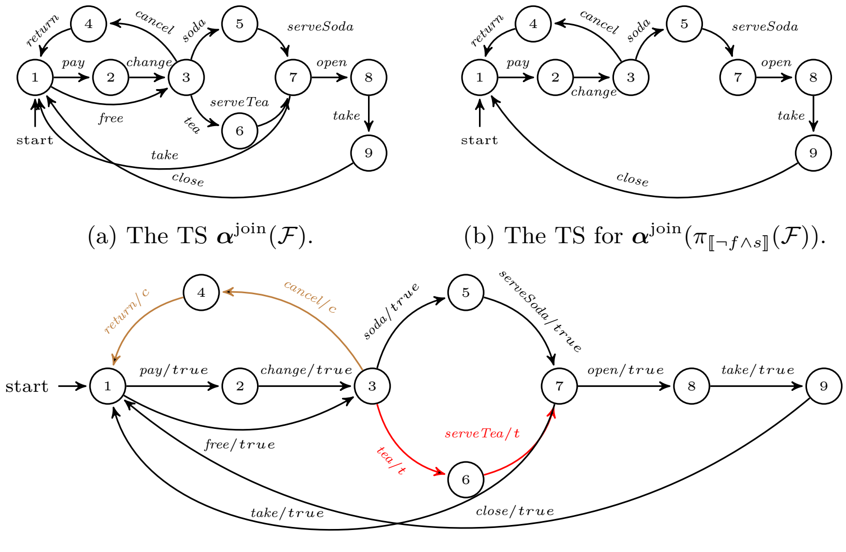
In the following, we will simply write (α, γ) for any Galois connection $\langle FeatExp(\mathbb{F})_{/\equiv}, \sqsubseteq\rangle \xleftrightarrow[\alpha]{\gamma} \langle FeatExp(\alpha(\mathbb{F}))_{/\equiv}, \sqsubseteq\rangle$ constructed using the operators presented in this section.

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 featured transition systems (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 (α, γ) 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 4. Consider the FTS $\mathcal{F}$ in Fig. 1 with the set of valid configurations $\mathbb{K} = \{\{v,s\},\{v,s,t,c,f\},\{v,s,c\},\{v,s,c,f\}\}$. We show $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F}), \boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg f \wedge s]\!]}(\mathcal{F}))$, and $\boldsymbol{\alpha}^{\text{fignore}}_{\{v,s,f\}}(\mathcal{F})$ in Fig. 2. Note that $\mathbb{K} \cap [\![\neg f \wedge s]\!] = \{\{v,s\},\{v,s,c\}\}$, and hence transitions annotated with the feature t (`Tea`) and f (`FreeDrinks`) are not present in $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg f \wedge s]\!]}(\mathcal{F}))$. Also note that in the case of $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F})$ and $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg f \wedge s]\!]}(\mathcal{F}))$ we obtain ordinary transition systems, since all transitions are labelled with the feature expression *true*.



(a) The TS $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F})$.

(b) The TS for $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg f \wedge s]\!]}(\mathcal{F}))$.

(c) The FTS for $\boldsymbol{\alpha}^{\text{fignore}}_{\{v,s\}}(\mathcal{F})$.

Fig. 2: Various abstractions of $\mathcal{F}$.

Property Preservation. We now show that abstracted FTSs have some interesting preservation properties.

Lemma 1. *Let $\chi, \psi_0, \psi_1, \ldots \in FeatExp(\mathbb{F})$, and $\mathbb{K}$ be a set of configurations over $\mathbb{F}$. 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 $i \geq 0$.*

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

It follows from Def. 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 $i = 1, \ldots, n$.*

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* $\alpha_1(\pi_{\mathbb{K}_1}(\mathcal{F})) \models [\alpha_1(\chi)]\phi \wedge \ldots \wedge \alpha_n(\pi_{\mathbb{K}_n}(\mathcal{F})) \models [\alpha_n(\chi)]\phi$*, 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 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.

4 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 systems. *f*PROMELA is designed for describing variational models; whereas TVL for describing the sets of features and configurations. We present *f*PROMELA and TVL and show their FTS semantics.

Syntax. *f*PROMELA is obtained from PROMELA [16] by adding *feature variables*, $\mathbb{F}$, and *guarded statements*. PROMELA is a non-deterministic modelling language designed for describing systems composed of concurrent processes that communicate asynchronously. A PROMELA program, P, consists of a finite set of processes to be executed concurrently. The basic statements of processes are given by:

$$\begin{array}{lll} \mathit{stm} & ::= & \texttt{skip} \;\mid\; \texttt{x := } \mathit{expr} \;\mid\; c?x \;\mid\; c!\mathit{expr} \;\mid\; \mathit{stm}_1 \texttt{ ; } \mathit{stm}_2 \;\mid \\ & & \texttt{if} :: g_1 \Rightarrow \mathit{stm}_1 \cdots :: g_n \Rightarrow \mathit{stm}_n :: \texttt{else} \Rightarrow \mathit{stm} \texttt{ fi} \;\mid \\ & & \texttt{do} :: g_1 \Rightarrow \mathit{stm}_1 \cdots :: g_n \Rightarrow \mathit{stm}_n \texttt{ od} \end{array}$$

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 features used in an *f*PROMELA program have to be declared as fields of the special type *features*. The new guarded statement introduced in *f*PROMELA is of the form: "`gd` :: $\psi_1 \Rightarrow \mathit{stm}_1 \ldots :: \psi_n \Rightarrow \mathit{stm}_n ::$ `else` $\Rightarrow \mathit{stm}$ `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 [18].)

TVL [5] is a textual modelling language for describing the set of all valid configurations, $\mathbb{K}$, for an *f*PROMELA program 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. 3 shows a simple *f*Promela program and the corresponding TVL model. After declaring feature variables in the *f*Promela program in Fig. 3a, a 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. 3b specifies four valid configurations: {`Main`}, {`Main`, `A`}, {`Main`, `B`}, {`Main`, `A`, `B`}. If we use the $\overline{\text{SNIP}}$ tool to check the assertion, $\mathtt{i} \geq 0$, in this example, we will obtain that it is satisfied by all (four) valid variants. If we include the constraint in comments in line 5 of Fig. 3b that excludes the variant: $\neg\mathtt{A} \wedge \neg\mathtt{B}$, then the assertion $\mathtt{i} > 0$ will also hold for all (three) valid variants.

```
typedef features {
  bool A; bool B;}
features f;
active proctype foo() {
  int i := 0;
  gd :: f.A ⇒ i++ :: else ⇒ skip dg;
  gd :: f.B ⇒ i++ :: else ⇒ skip dg;
  assert(i ≥ 0);
 }
```

(a) An *f*Promela program.

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

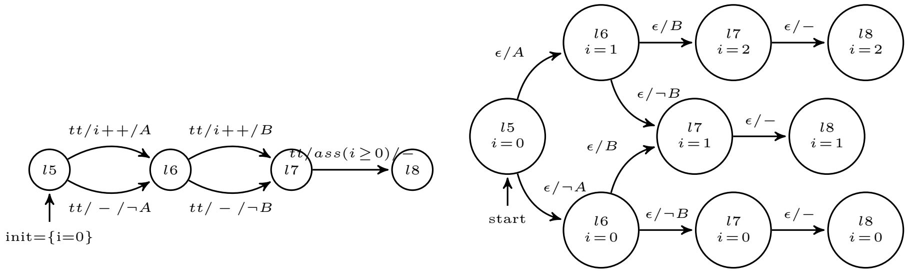
(b) A TVL model.

Fig. 3: A simple *f*Promela program and the corresponding TVL model

Semantics. We now show only the most relevant details of *f*Promela semantics. For the precise account of Promela semantics the reader is referred to [16]. Each *f*Promela program defines a so-called *featured program graph* (FPG), which formalizes the control flow of the program. The FPG represents a program graph [2] (or "finite state automaton" in [16]) in which transitions are explicitly linked with feature expressions. The vertices of the graph are control locations (represented by line numbers in the program) and its transition relation defines the control flow of the program. Each transition has *condition* under which it can be executed, an *effect* which specifies the 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 program. Let $Cond(V)$ denote the set of Boolean conditions over V, and $Assgn(V)$

denote all assignments over V. $\mathit{Eval}(V)$ is the set of all evaluations of V that assign concrete values to variables in V. A *featured program graph* over V and $\mathbb{F}$ is a tuple $(\mathit{Loc}, \mathit{tr}, \mathit{Loc}_0, \mathit{init}, \mathbb{K}, \mathit{fe})$, where Loc is a set of control locations, $\mathit{Loc}_0 \subseteq \mathit{Loc}$ is a set of initial locations, $\mathit{tr} \subseteq \mathit{Loc} \times \mathit{Cond}(V) \times \mathit{Assgn}(V) \times \mathit{Loc}$ is the transition relation, $\mathit{init} \in \mathit{Cond}(V)$ is the initial condition characterising the variables in the initial state, $\mathbb{K}$ is a set of configurations, and $\mathit{fe} : \mathit{trans} \to \mathit{FeatExp}(\mathbb{F})$ annotates transitions with feature expressions. The *semantics* of an FPG is an *FTS* obtained from "*unfolding*" the graph (see [2, Sect. 2] for details). The unfolded FTS is $(\mathit{Loc} \times \mathit{Eval}(V), \{\epsilon\}, \mathit{trans}, I, \mathit{Cond}(V), L, \mathbb{F}, \mathbb{K}, \delta)$, where the states are pairs of the form (l, v) for $l \in \mathit{Loc}, v \in \mathit{Eval}(V)$; action names are ignored (ϵ is an empty (dummy) action name); $I = \{(l, v) \mid l \in \mathit{Loc}_0, v \models \mathit{init}\}$; $L((l, v)) = \{g \in \mathit{Cond}(V) \mid v \models g\}$; and transitions are defined as: if $(l, g, a, l') \in \mathit{tr}$ and $v \models g$, then $((l, v), \epsilon, (l', \mathit{apply}(a, v))) \in \mathit{trans}$. Here, we write $v \models g$ if the evaluation v makes g *true*, and $\mathit{apply}(a, v)$ is the evaluation obtained after applying the assignment a to v. Given $t \in \mathit{trans}$, let $t' \in \mathit{tr}$ be the corresponding transition of the FPG. Then $\delta(t) = \mathit{true}$ if $\mathit{fe}(t')$ is undefined; and $\delta(t) = \mathit{fe}(t')$ otherwise. Hence, the semantics of an *f*Promela program follows the semantics of Promela, just adding feature expression from the FPG to the transitions. For example, in Fig 4 are shown the FPG and FTS for the family in Fig. 3.



(a) An FPG.

(b) An FTS.

Fig. 4: The semantics of the *f*Promela program in Fig. 3. Note that "lx" refers to the line number x from the program in Fig. 3a, and tt is short for *true*.

5 Variability Abstraction via Syntactic Transformation

We present the syntactic transformations of *f*Promela programs and TVL models introduced by projection and variability abstractions. Let P represent an *f*Promela program, 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 4.

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

Let (α, γ) be a Galois connection obtained from our calculus in Section 3. The abstract $\alpha(P)$ and $\alpha(T)$ are obtained by defining a translation recursively over the structure of α. The function α 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 $\alpha(\mathit{stm}_1;\mathit{stm}_2) = \alpha(\mathit{stm}_1);\alpha(\mathit{stm}_2)$. We discuss the rewrites for "gd" below.

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

$$\begin{aligned}
&\boldsymbol{\alpha}^{\text{join}}(\texttt{gd} :: \psi_1 \Rightarrow \mathit{stm}_1 \ldots :: \psi_n \Rightarrow \mathit{stm}_n :: \texttt{else} \Rightarrow \mathit{stm}'\, \texttt{dg}) = \\
&\quad \texttt{if} :: \boldsymbol{\alpha}^{\text{join}}(\psi_1) \Rightarrow \boldsymbol{\alpha}^{\text{join}}(\mathit{stm}_1) \ldots :: \boldsymbol{\alpha}^{\text{join}}(\psi_n) \Rightarrow \boldsymbol{\alpha}^{\text{join}}(\mathit{stm}_n) \\
&\qquad :: \boldsymbol{\alpha}^{\text{join}}(\neg(\psi_1 \vee \ldots \psi_n)) \Rightarrow \boldsymbol{\alpha}^{\text{join}}(\mathit{stm}')\, \texttt{fi}
\end{aligned}$$

For $\boldsymbol{\alpha}_A^{\text{fignore}}$, the transformation is

$$\begin{aligned}
&\boldsymbol{\alpha}_A^{\text{fignore}}(\texttt{gd} :: \psi_1 \Rightarrow \mathit{stm}_1 \ldots :: \psi_n \Rightarrow \mathit{stm}_n :: \texttt{else} \Rightarrow \mathit{stm}'\, \texttt{dg}) = \\
&\texttt{gd} :: \boldsymbol{\alpha}_A^{\text{fignore}}(\psi_1) \Rightarrow \boldsymbol{\alpha}_A^{\text{fignore}}(\mathit{stm}_1) \ldots :: \boldsymbol{\alpha}_A^{\text{fignore}}(\neg(\psi_1 \vee \ldots \psi_n)) \Rightarrow \boldsymbol{\alpha}_A^{\text{fignore}}(\mathit{stm}')\texttt{dg}
\end{aligned}$$

and the feature A is removed from T obtaining a new $\boldsymbol{\alpha}_A^{\text{fignore}}(T)$, when $\mathbb{F} \backslash \{A\} \neq \emptyset$. Otherwise, if $\mathbb{F} \backslash \{A\} = \emptyset$, then $\boldsymbol{\alpha}_A^{\text{fignore}}(P)$ is a Promela program and $\boldsymbol{\alpha}_A^{\text{fignore}}(T)$ is empty.

For $\alpha_2 \circ \alpha_1$, we have $\alpha_2 \circ \alpha_1(\texttt{gd} :: \psi_1 \Rightarrow \mathit{stm}_1 \ldots \texttt{dg}) = \alpha_2(\alpha_1(\texttt{gd} :: \psi_1 \Rightarrow \mathit{stm}_1 \ldots \texttt{dg}))$. Similarly, we transform the TVL model T.

Theorem 4. *Let P and T be an f*Promela *program and the corresponding TVL model, and (α, γ) be a Galois connection. We have: $\alpha([\![P]\!]_T) = [\![\alpha(P)]\!]_{\alpha(T)}$.*

6 Evaluation

We now evaluate our variability abstractions. First, we show how variability abstractions can render analysis of previously infeasible model families, feasible. Second, we turn to the main point of this paper: That instead of verifying properties using a family-based model checker (e.g., $\overline{\text{SNIP}}$), we can use variability abstraction to obtain an abstracted family-of-models (with a low number of variants) that can then be model checked using a single-system model checker (e.g., SPIN). By soundness of abstraction, *if* we are able to verify properties on the abstracted model family, we may safely conclude that they also hold on the original (unabstracted) model family. We investigate improvements in *performance* (Time) and *memory consumption* (Space) on the MinePump family-model [7] that comes with the installation of $\overline{\text{SNIP}}$. Finally, we do a case study on the MinePump. We show how various variability abstractions may be tailored for analysis of properties of the MinePump.

All of our abstractions are applied using our *f*Promela Reconfigurator (model-family-*to*-model-family) transformation tool[3] as described in Section 5. All

[3] The *f*Promela Reconfigurator tool (including all benchmarks) is available from: [http://ahmadsalim.github.io/p3-tool].

	unabstracted			*abstracted*			*improvement*	
$\lvert\mathbb{F}\rvert$	$\lvert\mathbb{K}\rvert$	Time	Space	$\lvert\alpha(\mathbb{K})\rvert$	Time	Space	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	— n/a —	

Fig. 5: Verifying deadlock absence in MinePump for increasing levels of variability (without vs. with maximal abstraction, $\alpha = \boldsymbol{\alpha}^{\text{join}}$, confounding all configurations).

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.

***A Characterization of* MinePump.** The *f*Promela MinePump model 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`.

From Infeasible to Feasible Analysis via Abstraction. 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 quickly becomes infeasible. They take too long time to analyze.

Let us for a moment focus on (single-system) model checkers which may be applied at the family level by "brute force" model checking *all* variants of a given model family, one by one. As an experiment, we gradually added variability to the family-model in Figure 3. Already for $|\mathbb{F}| = 11$ (for which $|\mathbb{K}| = 2^{11} = 2,048$ variants), analysis time to check the assertion becomes almost a minute. For $|\mathbb{F}| = 25$, analysis time ascends to almost a year. 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.

Family-Based Model Checking without a Family-Based Model Checker. Recently, researchers have introduced *family-based model-checking* [6] that work at the family level and thus do not explicitly check all variants, one by one. (Analogous endeavors have been undertaken in, for instance, type checking [18], and dataflow analysis [3].) Much effort has been dedicated to speeding up analyses via *improving representation*; in particular, by exploiting information that may be "shared" among multiple configurations via BDDs. In this paper, we propose to speed up analyses via *increasing abstraction* on the configuration space. In fact, increasing abstraction and improving representation are orthogonal; i.e., they may cooperatively speed up analyses even further!

Figure 5 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). We verify the deadlock freedom property. *Improvement* is the relative comparison of *unabstracted* vs. *abstracted*. TIME is the time to model-check (in seconds) and SPACE is the number of explored states plus the number of re-explored states (which 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 program, 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 [6,7].

The rows of Figure 5 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, preseving 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 analysis the *abstracted* system in 0.09 seconds (exploring $170,670$ transitions).

Devising Abstractions for Properties (A Case Study of* MINEPUMP*). We start by considering four universal properties, φ_1 to φ_4 (taken from [7]), that are intended to be satisfied by all variants, listed in Figure 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

Φ	*property*
(φ_0)	$(\mathtt{GF\,readCommand}) \wedge (\mathtt{GF\,readAlarm}) \wedge (\mathtt{GF\,readLevel})$ *Fairness: The system will infinitely often read messages of various types.*
φ_1	*Absence of deadlock.*
φ_2	$\mathtt{G}\,(\neg\mathtt{pumpOn} \vee \mathtt{stateRunning})$ *If the pump is switched on, then the controller state is "running".*
φ_3	$\varphi_0 \Rightarrow (\neg\mathtt{GF}\,(\neg\mathtt{pumpOn} \wedge \neg\mathtt{methane} \wedge \mathtt{highWater}))$ *Assuming fairness (φ_0), the pump is never indefinitely off* *when the water level is high and there is no methane.*
φ_4	$\mathtt{G}\,((\neg\mathtt{pumpOn} \wedge \mathtt{lowWater} \wedge \mathtt{F\,highWater}) \Rightarrow (\neg\mathtt{pumpOn}\ \mathtt{U}\ \mathtt{highWater}))$ *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.*
φ_5	$\neg(\mathtt{GF\,pumpOn})$ *The pump cannot be switched on infinitely often.*
φ_6	$\varphi_0 \Rightarrow \neg\mathtt{FG}\,(\mathtt{pumpOn} \wedge \mathtt{methane})$ *Assuming fairness (φ_0), the system cannot be in a state* *where the pump is on indefinitely in the presence of methane.*

Fig. 6: Properties for the MinePump (taken from [7]).

prop-	*unabstracted*			*abstracted*			*improvement*	
-erty	$\lvert\mathbb{K}\rvert$	Time	Space	$\lvert\alpha(\mathbb{K})\rvert$	Time	Space	Time	Space
φ_1	128	2.96 s	251 k	1	0.04 s	34 k	74 ×	7.4 ×
φ_2	128	4.28 s	326 k	1	0.05 s	34 k	86 ×	9.6 ×
φ_3	128	6.37 s	441 k	1	0.09 s	161 k	71 ×	2.7 ×
φ_4	128	5.98 s	420 k	1	0.05 s	57 k	120 ×	7.3 ×
φ_5	128	3.20 s	207 k	3	0.11 s	12 k	29 ×	16.6 ×
φ_6	128	4.54 s	309 k	4	0.16 s	42 k	28 ×	7.3 ×

Fig. 7: Verification of above MinePump properties using tailored abstractions.

only one configuration, $|\boldsymbol{\alpha}^{\text{join}}(\mathbb{K}_{\text{MinePump}})| = 1$. The first four rows of Figure 7 organizes the results of maximally abstracting the MinePump prior to verfication of properties, φ_1 to φ_4. Consistent with our expectations and previous results (cf. Figure 5), maximal abstraction translates to massive improvements in both Time and Space on a family-model with many variants ($|\mathbb{K}| = 128$). In fact, model checking is between 71 and 120 times faster.

We now consider non-universal properties which are preserved by some variants and violated by others: φ_5 and φ_6 (see Figure 6). Property φ_5 (concerning the pump being switched on), is violated by all variants, 32 in total, for which $\mathtt{Start} \wedge \mathtt{High}$ 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: $\pi_{\mathtt{Start}\wedge\mathtt{High}}$ (corresponding to the above 32 configurations), $\pi_{\neg\mathtt{Start}}$ (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 φ_5 on each of them. For the first abstracted projection, $\boldsymbol{\alpha}^{\text{join}}(\pi_{\texttt{Start} \wedge \texttt{High}}(\textsc{MinePump}))$, 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 φ_5 is *satisfied*.

Overall, we can see that our approach is significantly faster. The second-last row of Figure 7 shows that analysis time drops from 3.20 seconds when 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, φ_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). Of course, much of the performance improvement is due to the highly-optimized industry-strength SPIN tool (compared to the $\overline{\text{SNIP}}$ research prototype). Previous work attributes a factor of two advantage for (brute force) SPIN over $\overline{\text{SNIP}}$ [7]. However, for models with more variability (larger values of $|\mathbb{F}|$), a constant factor will be dwarfed by the inherent exponential blow up.

We can also use $\boldsymbol{\alpha}^{\text{fignore}}$ abstraction to speed up the family-based model checker. For the property φ_5, we call $\overline{\text{SNIP}}$ on $\boldsymbol{\alpha}^{\text{fignore}}_{\mathbb{F}\setminus\{\texttt{Start},\texttt{High}\}}(\textsc{MinePump})$, and we obtain the same counter-examples as in the unabstracted case for the variants in $\texttt{Start} \wedge \texttt{High}$. However, the verification time is reduced from 3.20 to 0.97 sec, and the number of examined transitions is reduced from $207,377$ to $54,376$.

In conclusion, by exploiting high-level 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.

7 Related Work

Abstractions for family-based model checking. *Simulation-based abstraction* of family-based model checking was introduced in [9]. 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 model 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 [9] 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.

Variability-aware abstraction procedures based on counterexample guided abstraction refinement (CEGAR) have been proposed in [10]. Abstractions are introduced by using existential F-abstraction functions, and simulation relation is used to relate different abstraction levels. Three types of 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 [10] 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 [10] 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 [10] are applied on feature program graphs, we apply our abstractions as preprocessor transformations directly on high-level programs thus avoiding to generate any intermediate concrete model in the memory.

Family-based Static Analysis. 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 type checking [18], lifted data-flow analysis [3], lifted model checking [6,7], etc.

A formal methodology for systematic derivation of lifted data-flow analyses for program families with `#ifdef`-s is proposed in [19]. The method uses the calculational approach to abstract interpretation of Cousot [11] in order to derive a directly operational lifted analysis. In [14], we devise an expressive calculus of variability abstractions that are used for deriving abstracted lifted data-flow analyses. Such variability abstractions enable deliberate trading of precision for speed in lifted analysis. Hence, they tame the exponential blow-up caused by the large number of features and variants in an program family. In this work, we pursue this line of work by adapting variability abstractions in the context of lifted model checking. However, the abstractions in [14] are directed at reducing the configuration space $|\mathbb{K}|$ since the elements of the property domain are $|\mathbb{K}|$-sized tuples. On the other hand, the abstractions defined here aim at reducing the space of feature expressions since the variability-sensitive information in FTSs, fLTL formulae, and *f*PROMELA programs is encoded by using feature expressions.

8 Conclusion

We have proposed variability abstractions to derive abstract model checking for families of related systems. The abstractions are applied before model generation directly on *f*PROMELA programs. The evaluation confirms that interesting properties can be efficiently verified in this way by only a few calls to SPIN. The approach is well suited for properties that are either satisfied by all variants, or unsatisfied for small number of features.

References

1. APEL, S., BATORY, D. S., KÄSTNER, C., AND SAAKE, G. *Feature-Oriented Software Product Lines - Concepts and Implementation.* Springer, 2013.

2. Baier, C., and Katoen, J. *Principles of model checking.* MIT Press, 2008.
3. Brabrand, C., Ribeiro, M., Tolêdo, T., and Borba, P. Intraprocedural dataflow analysis for software product lines. In *Proceedings of the 11th International Conference on Aspect-oriented Software Development, AOSD 2012* (2012), R. Hirschfeld, É. Tanter, K. J. Sullivan, and R. P. Gabriel, Eds., ACM, pp. 13–24.
4. Clarke, E. M., Grumberg, O., and Long, D. E. Model checking and abstraction. *ACM Trans. Program. Lang. Syst. 16*, 5 (1994), 1512–1542.
5. Classen, A., Boucher, Q., and Heymans, P. A text-based approach to feature modelling: Syntax and semantics of TVL. *Sci. Comput. Program. 76*, 12 (2011), 1130–1143.
6. Classen, A., Cordy, M., Heymans, P., Legay, A., and Schobbens, P. Model checking software product lines with SNIP. *STTT 14*, 5 (2012), 589–612.
7. Classen, A., Cordy, M., Schobbens, P., Heymans, P., Legay, A., and Raskin, J. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. *IEEE Trans. Software Eng. 39*, 8 (2013), 1069–1089.
8. Clements, P., and Northrop, L. *Software Product Lines: Practices and Patterns.* Addison-Wesley, 2001.
9. Cordy, M., Classen, A., Perrouin, G., Schobbens, P., Heymans, P., and Legay, A. Simulation-based abstractions for software product-line model checking. In *34th International Conference on Software Engineering, ICSE 2012* (2012), M. Glinz, G. C. Murphy, and M. Pezzè, Eds., IEEE, pp. 672–682.
10. Cordy, M., Heymans, P., Legay, A., Schobbens, P., Dawagne, B., and Leucker, M. 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)* (2014), S. Cheung, A. Orso, and M. D. Storey, Eds., ACM, pp. 190–201.
11. Cousot, P. The calculational design of a generic abstract interpreter. In *Calculational System Design*, M. Broy and R. Steinbrüggen, Eds. NATO ASI Series F. IOS Press, Amsterdam, 1999.
12. Czarnecki, K., and Antkiewicz, M. Mapping features to models: A template approach based on superimposed variants. In *Generative Programming and Component Engineering, 4th Int. Conf., GPCE 2005* (2005), vol. 3676 of *LNCS*, pp. 422–437.
13. Dams, D., Gerth, R., and Grumberg, O. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst. 19*, 2 (1997), 253–291.
14. Dimovski, A., Brabrand, C., and Wąsowski, A. Variability abstractions: Trading precision for speed in family-based analyses. In *ECOOP 2015 - Object-Oriented Programming - 29th European Conference* (2015), p. To appear.
15. Gallardo, M., Martínez, J., Merino, P., and Pimentel, E. aspin: A tool for abstract model checking. *STTT 5*, 2-3 (2004), 165–184.
16. Holzmann, G. J. *The SPIN Model Checker - primer and reference manual.* Addison-Wesley, 2004.
17. Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. Feature-Oriented Domain Analysis (FODA) feasibility study. Tech. rep., Carnegie-Mellon University Software Engineering Institute, November 1990.
18. Kästner, C., and Apel, S. Type-checking software product lines - A formal approach. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2008)* (2008), IEEE, pp. 258–267.
19. Midtgaard, J., Brabrand, C., and Wasowski, A. Systematic derivation of static analyses for software product lines. In *13th Int. Conf. on MODULARITY '14* (2014), pp. 181–192.

Note to the reviewers:
The remaining part of the document contains supplementary material purely for your convenience, to aid the evaluation process. It presents no new results, and is not intended to be published in the conference proceedings.

A Proofs

Proof (Theorem 1). Let $\psi \in FeatExp(\mathbb{F})$ and $\varphi' \in FeatExp(\emptyset)$. We have:

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

□

Proof (Theorem 2). Let $\psi \in FeatExp(\mathbb{F})$, and $\varphi' \in FeatExp(\mathbb{F} \backslash \{A\})$ be in NNF.

$$
\begin{aligned}
&\boldsymbol{\alpha}_A^{\text{fignore}}(\psi) \models \varphi' \\
&\iff \psi[l_A \mapsto true] \models \varphi' && \text{(by def. of } \boldsymbol{\alpha}_A^{\text{fignore}}\text{)} \\
&\iff \psi \models (\varphi' \wedge A) \vee (\varphi' \wedge \neg A) && \text{(by Lemma 2 in App. A)} \\
&\iff \psi \models \boldsymbol{\gamma}_A^{\text{fignore}}(\varphi') && \text{(by def. of } \boldsymbol{\gamma}_A^{\text{fignore}}\text{)}
\end{aligned}
$$

□

Lemma 2. *Let ψ and φ' 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 ψ.

Case $\psi = A$: Consider the 'if' direction. By assumption, we have $true \models \varphi'$, which means that φ' is valid. Then $A \models (\varphi' \wedge A)$, so it follows $A \models (\varphi' \wedge A) \vee (\varphi' \wedge \neg A)$. The opposite direction is similar.

Case $\psi = \neg A$: is analogous to the previous one.

Case $\psi = \psi_1 \wedge \psi_2$: Consider the 'if' direction. By assumption, we have $\psi_1[l_A \mapsto true] \models \varphi'$ and $\psi_2[l_A \mapsto true] \models \varphi'$. By IH, we have $\psi_1 \models (\varphi' \wedge A) \vee (\varphi' \wedge \neg A)$ and $\psi_2 \models (\varphi' \wedge A) \vee (\varphi' \wedge \neg A)$. It follows $\psi_1 \wedge \psi_2 \models (\varphi' \wedge A) \vee (\varphi' \wedge \neg A)$. The opposite direction is similar.

Case $\psi = \psi_1 \vee \psi_2$: Consider the 'if' direction. By assumption, we have $\psi_1[l_A \mapsto \mathit{true}] \models \varphi'$ and $\psi_2[l_A \mapsto \mathit{true}] \models \varphi'$. By IH, we have $\psi_1 \models (\varphi' \wedge A) \vee (\varphi' \wedge \neg A)$ and $\psi_2 \models (\varphi' \wedge A) \vee (\varphi' \wedge \neg A)$. It follows $\psi_1 \vee \psi_2 \models (\varphi' \wedge A) \vee (\varphi' \wedge \neg A)$. The opposite direction is similar.

□

Proof (Lemma 1). By induction on the structure of α.

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

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

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

□

Proof (Theorem 3). We proceed by contraposition. Suppose that $\mathcal{F} \not\models [\chi]\phi$. 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 ρ, $t_i = s_i \xrightarrow{\lambda_{i+1}} s_{i+1}$ for $i = 0, 1, \ldots$, we have that $k \models \delta(t_i)$ for all $i \geq 0$. By Lemma 1, 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 ρ is realizable for $\alpha(\mathcal{F})$, i.e. $\rho \in [\![\pi_{k'}(\alpha(\mathcal{F}))]\!]_{TS}$ and $\rho \models \neg\phi$. It follows that $\alpha(\mathcal{F}) \not\models [\alpha(\chi)]\phi$. □

Proof (Theorem 4). 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 α are applied on feature expressions, which can be introduced in FTSs only through "`gd`"-s. Thus, it is the same wether 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. □