Variability-specific Abstraction Refinement for Family-based Model Checking [⋆]

Aleksandar S. Dimovski and Andrzej Wąsowski

Computer Science, IT University of Copenhagen, Denmark

Abstract. Variational systems are ubiquitous in many application areas today. They use features to control presence and absence of system functionality. One challenge in the development of variational systems is their formal analysis and verification. Researchers have addressed this problem by designing aggregate so-called family-based verification algorithms. Family-based model checking allows simultaneous verification of all variants of a system family (variational system) in a single run by exploiting the commonalities between the variants. Yet, the computational cost of family-based model checking still greatly depends on the number of variants. In order to make it computationally cheaper, we can use variability abstractions for deriving abstract family-based model checking, where the variational model of a system family is replaced with an abstract (smaller) version of it which preserves the satisfaction of LTL properties. The variability abstractions can be combined with different partitionings of the set of variants to infer various verification scenarios for the variational model. However, manually finding an optimal verification scenario is hard since it requires a good knowledge of the family and property, while the number of possible scenarios is very large.
In this work, we present an automatic iterative abstraction refinement procedure for family-based model checking. We use Craig interpolation to refine abstract variational models based on the obtained spurious counterexamples (traces). The refinement procedure works until a genuine counterexample is found or the property satisfaction is shown for all variants in the family. We illustrate the practicality of this approach for several variational benchmark models.

1 Introduction

Software Product Line Engineering (SPLE) [9] is a popular methodology for building a family of related systems. A large number of related systems (*variants*) are developed by systematically reusing common parts. Each variant is specified in terms of *features* (statically configured options) selected for that particular variant. Due to the popularity of SPLs in embedded and critical system domain (e.g. cars, phones, avionics), they require rigourous verification and analysis.

[⋆] Partially supported by The Danish Council for Independent Research under a Sapere Aude project, VARIETE.

Model checking is a well-known technique for automatic verification of systems against properties expressed in temporal logic [1]. Model checking families of systems is more difficult than model checking single systems, since the number of possible variants is exponential in the number of features. Hence, the simplest enumerative variant-by-variant approach, that applies single-system model checking to each individual variant of a system family, is very inefficient. Indeed, a given execution behaviour is checked as many times as the number of variants that are able to execute it. In order to address this problem, new dedicated family-based model checking algorithms have been introduced [8,7,10]. They rely on using compact mathematical structures (so called variational models or featured transition systems) for modelling variational systems, which take the commonality within the family into account, and on which specialized family-based (variability-aware) model checking algorithms can be applied. Each execution behaviour in a variational model is associated with the exact set of variants able to produce it. Therefore, the family-based algorithms check an execution behaviour only once, regardless of how many variants can produce it. In this way, they are able to model check all variants of a family simultaneously in a single step and pinpoint those variants that violate properties. In order to further speed-up family-based model checking, a range of variability abstractions can be introduced [13,14]. They give rise to abstract family-based model checking. The abstractions are applied at the variability level and aim to reduce the exponential blowup of the number of configurations (variants) to something more tractable by manipulating the configuration space of the family. Abstractions can be combined with partitionings of the set of all variants to generate various verification scenarios. Still, suitable verification scenarios are currently chosen manually from a large set of possible combinations. This often requires a user to have a considerable knowledge of a variational system and property. In order for this approach to be used more widely in industry, automatic techniques are needed for generating verification scenarios.

Abstraction refinement [4,5,10] has proved to be one of the most effective techniques for automatic verification of systems with very large state spaces. In this paper, we introduce a purely variability-specific (state-independent) approach to abstraction refinement, which is used for automatic verification of LTL properties over variational models. In general, each variability abstraction computes an over-approximation of the original model, in a such a way that if some property holds for the smaller abstract model then it will hold for the original one. However, if the property does not hold in the abstract model, the found counterexample may be the result of some behaviour in the over-approximation which is not present in the original model. In this case, it is necessary to refine the abstraction so that the behaviour which caused the spurious counterexample is eliminated. The verification procedure starts with the coarsest variability abstraction, and then the obtained abstract model is fed to a model checker. If no counterexample is found, then all variants satisfy the given property. Otherwise, the counterexamples are analysed and classified as either *genuine*, which correspond to execution behaviours of some variants in the original model, or *spurious*, which are introduced due to the abstraction. If a genuine counterexample exist, the corresponding variants do not satisfy the given property; otherwise a spurious counterexample is used to refine the abstract models. The procedure is then repeated on the refined abstract variational model only for variants for which no conclusive results have been found. We use Craig interpolation [18,27] to extract from a spurious counterexample (i.e. the unsatisfiable feature expression associated with it) the relevant information which needs to be known in order to show the unsatisfiability of the associated feature expression. This information is used to compute refined abstract models for the next iteration. The main contribution of this paper is an efficient automatic abstraction refinement procedure for family-based model checking, which uses variability-aware information obtained from spurious counterexamples to guide the verification process. When the employed variability abstractions give rise to abstract models verifiable by a single-system model checker, we obtain a completely automatic alternative to a dedicated family-based model checker. The experiments show that the proposed abstraction refinement procedure combined with the single-system model checker $\underline{\text{SPIN}}$ achieves performance gains compared to the family-based model checker $\overline{\text{SNIP}}$ when applied to several benchmark variational systems for some interesting properties.

2 Abstract family-based model checking

We now introduce featured transition systems (FTSs) [8] for modelling variational systems, fLTL temporal formulae [8] for specifying properties of variational systems, and variability abstractions [13,14] for defining abstract FTSs.

2.1 Featured transition systems

Let $\mathbb{F} = \{A_1, \ldots, A_n\}$ be a finite set of Boolean variables representing the features available in a variational system. A specific subset of features, $k \subseteq \mathbb{F}$, known as *configuration*, specifies a *variant* (valid product) of a variational system. The *set of all valid configurations* (variants) is defined as: $\mathbb{K} \subseteq 2^{\mathbb{F}}$. An alternative representation of configurations is based upon propositional formulae. Each configuration $k \in \mathbb{K}$ can be represented by a formula: $k(A_1) \wedge \ldots \wedge k(A_n)$, where $k(A_i) = A_i$ if $A_i \in k$, and $k(A_i) = \neg A_i$ if $A_i \notin k$ for $1 \leq i \leq n$. We will use both representations interchangeably. The set of valid configurations is typically described by a feature model [22], but in this work we disregard syntactic representations of the set $\mathbb{K}$.

The behaviour of individual variants is given with transition systems.

Definition 1. *A transition system (TS) is a tuple $\mathcal{T} = (S, Act, trans, I, AP, L)$, where S is a set of states; Act is a set of actions; $trans \subseteq S \times Act \times S$ is a transition relation* [1]*; $I \subseteq S$ is a set of initial states; AP is a set of atomic propositions; and $L : S \to 2^{AP}$ is a labelling function.*

[1] We often write $s_1 \xrightarrow{\lambda} s_2$ when $(s_1, \lambda, s_2) \in trans$.

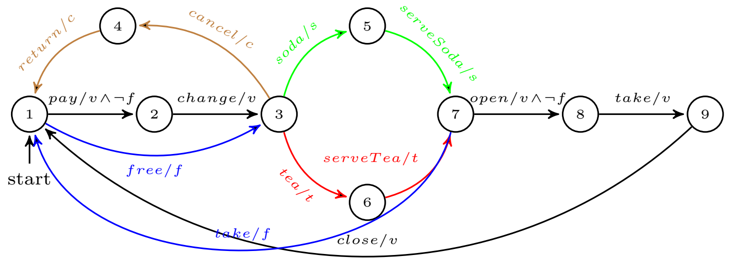
– *An* execution *(behaviour) of $\mathcal{T}$ is a nonempty, infinite sequence $\rho = s_0\lambda_1 s_1\lambda_2\dots$ with $s_0 \in I$ such that $s_i \xrightarrow{\lambda_{i+1}} s_{i+1}$ for all $i \geq 0$. The* semantics *of the TS $\mathcal{T}$, denoted as $[\![\mathcal{T}]\!]_{TS}$, is the set of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

The combined behaviour of a whole system family is compactly represented with *featured transition systems* [8]. They are TSs where transitions are also labelled with feature expressions, $FeatExp(\mathbb{F})$, which represent propositional logic formulae defined over $\mathbb{F}$ as: $\psi ::= true \mid A \in \mathbb{F} \mid \neg\psi \mid \psi_1 \wedge \psi_2$. The feature expression $\psi \in FeatExp(\mathbb{F})$ indicates for which variants the corresponding transition is enabled.

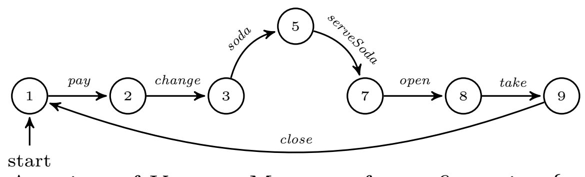
Definition 2. *An featured transition system (FTS) represents a tuple $\mathcal{F} = (S, Act, trans, I, AP, L, \mathbb{F}, \mathbb{K}, \delta)$, where $S, Act, trans, I, AP$, and L are defined as in TS; $\mathbb{F}$ is the set of available features; $\mathbb{K}$ is a set of valid configurations; and $\delta : trans \to FeatExp(\mathbb{F})$ is a total function labelling transitions with feature expressions. We write $[\![\delta(t)]\!]$ to denote the set of variants that satisfy $\delta(t)$, i.e. $k \in [\![\delta(t)]\!]$ iff $k \models \delta(t)$. Moreover:*

– *The* projection *of an FTS $\mathcal{F}$ to a variant $k \in \mathbb{K}$, denoted as $\pi_k(\mathcal{F})$, is the TS $(S, Act, trans', I, AP, L)$, where $trans' = \{t \in trans \mid k \models \delta(t)\}$.*
– *The* projection *of an FTS $\mathcal{F}$ to a set of varaints $\mathbb{K}' \subseteq \mathbb{K}$, denoted as $\pi_{\mathbb{K}'}(\mathcal{F})$, is the FTS $(S, Act, trans', I, AP, L, \mathbb{F}, \mathbb{K}', \delta)$, where $trans' = \{t \in trans \mid \exists k \in \mathbb{K}'.k \models \delta(t)\}$.*
– *The* semantics *of an FTS $\mathcal{F}$, denoted as $[\![\mathcal{F}]\!]_{FTS}$, is the union of behaviours of the projections on all variants $k \in \mathbb{K}$, i.e. $[\![\mathcal{F}]\!]_{FTS} = \cup_{k\in\mathbb{K}}[\![\pi_k(\mathcal{F})]\!]_{TS}$.*
– *The* size *of an FTS $\mathcal{F}$ is defined as [8]: $|\mathcal{F}| = |S| + |trans| + |expr| + |\mathbb{K}|$, where $|expr|$ is the size of all feature expressions bounded by $O(2^{|\mathbb{F}|} \cdot |trans|)$.*

Example 1. Throughout this paper, we will use a beverage vending machine as a running example [8]. The VendingMachine family has five features: `VendingMachine` (denoted by v) for purchasing a drink which is a mandatory root feature enabled in all products; `Tea` (denoted by t) for serving tea; `Soda` (denoted by s) for serving soda; `CancelPurchase` (denoted by c) for canceling a purchase after a coin is entered; and `FreeDrinks` (denoted by f) for offering free drinks. The FTS of VendingMachine is shown in Fig. 1a. The feature expression label of a transition is shown next to its label action, separated by a slash. The transitions enabled by the same feature are colored in the same way. For example, the transition ③ $\xrightarrow{soda/s}$ ⑤ is enabled for variants that contain the feature s. By combining various features, a number of variants of this VendingMachine can be obtained. In Fig. 1b is shown the basic version of VendingMachine that only serves soda, which is described by the configuration: $\{v, s\}$ (or, as formula $v \wedge s \wedge \neg t \wedge \neg c \wedge \neg f$). It takes a coin, returns change, serves soda, opens a compartment so that the customer can take the soda, before closing it again. We can obtain the basic vending machine in Fig. 1b by projecting the FTS in Fig. 1a to the configuration $\{v, s\}$. The set of all valid configurations of VendingMachine can be obtained by combining the above features. For example, we can have $\mathbb{K} = \{\{v, s\}, \{v, s, t, c, f\}, \{v, s, c\}, \{v, s, c, f\}\}$. □



(a) The FTS for VendingMachine.



(b) A variant of VendingMachine for configuration $\{v, s\}$.

Fig. 1: The VendingMachine variational system.

2.2 fLTL Properties

The model checking problem consists of determining whether a model satisfies a given property expressed as LTL (linear time logic) temporal formula [1].

Definition 3. *An LTL formula ϕ is defined as: $\phi ::= true \mid a \in AP \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \bigcirc\phi \mid \phi_1 U \phi_2$.*

– *Satisfaction of a formula ϕ 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 ρ) is defined as:*

$$\begin{aligned}
&\rho \models true, \qquad \rho \models a \quad \textit{iff} \quad a \in L(s_0),\\
&\rho \models \neg\phi \quad \textit{iff} \quad \rho \not\models \phi, \qquad \rho \models \phi_1 \wedge \phi_2 \quad \textit{iff} \quad \rho \models \phi_1 \textit{ and } \rho \models \phi_2,\\
&\rho \models \bigcirc\phi \quad \textit{iff} \quad \rho_1 \models \phi\\
&\rho \models \phi_1 U \phi_2 \quad \textit{iff} \quad \exists k \geq 0.\, \rho_k \models \phi_2 \textit{ and } \forall j \in \{0,\ldots,k-1\}.\, \rho_j \models \phi_1
\end{aligned}$$

– *A TS $\mathcal{T}$ satisfies a formula ϕ, denoted as $\mathcal{T} \models \phi$, iff $\forall \rho \in [\![\mathcal{T}]\!]_{TS}.\, \rho \models \phi$.*

Note that other temporal operators can be defined as well: $\Diamond\phi = true\, U \phi$ (eventually) and $\Box\phi = \neg\Diamond\neg\phi$ (always). When we consider variational systems, we sometimes want to define properties with a modality that specifies the set of variants for which they hold.

Definition 4. – *An feature LTL (fLTL) formula is defined as: $[\chi]\phi$, where ϕ is an LTL formula and $\chi \in FeatExp(\mathbb{F})$ is a feature expression.*

– *An FTS $\mathcal{F}$ satisfies an fLTL formula $[\chi]\phi$, denoted as $\mathcal{F} \models [\chi]\phi$, iff $\forall k \in \mathbb{K} \cap [\![\chi]\!].\, \pi_k(\mathcal{F}) \models \phi$. An FTS $\mathcal{F}$ satisfies an LTL formula ϕ iff $\mathcal{F} \models [true]\phi$.*

Note that $\mathcal{F} \models [\chi]\phi$ iff $\pi_{[\![\chi]\!]}(\mathcal{F}) \models \phi$. Therefore, for simplicity in the following we focus on verifying only LTL properties ϕ.

Example 2. Consider the FTS VendingMachine in Fig. 1a. Suppose that states ⑤ and ⑥ are labelled with the proposition `selected`, and the state ⑧ with the proposition `open`. An example property ϕ is: $\Box(\texttt{selected} \implies \Diamond\texttt{open})$, which states that after selecting a beverage, the machine will eventually `open` the compartment to allow the customer to take his drink. The basic vending machine satisfies this property: $\pi_{\{v,s\}}(\textsc{VendingMachine}) \models \phi$, but the entire variational system does not satisfy it: $\textsc{VendingMachine} \not\models \phi$. For example, if the feature f (`FreeDrinks`) is enabled, a counter-example where the state ⑧ is never reached is: ① → ③ → ⑤ → ⑦ → ① → .... The set of violating products is $\{\{v, s, t, c, f\}, \{v, s, c, f\}\} \subseteq \mathbb{K}$. However, we have that $\textsc{VendingMachine} \models [\neg f]\phi$. Therefore, we can conclude that the feature f is responsible for violation of the property ϕ. $\Box$

2.3 Variability abstractions

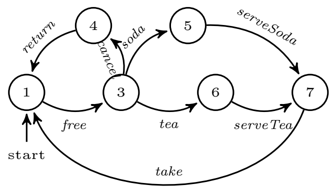
We now define variability abstractions [13,14] for decreasing the sizes of FTSs, in particular for reducing the number of features, the configuration space, and the size of feature expressions. The goal of variability abstractions is to weaken feature expressions, in order to make transitions in FTSs available to more variants. We define variability abstractions as Galois connections for reducing the Boolean complete 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 $\psi \in \mathit{FeatExp}(\mathbb{F})$ 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 $false$, and the greatest element is *true*. Subsequently, we will lift the definition of variability abstractions to FTSs.

The *join abstraction*, $\boldsymbol{\alpha}^{\text{join}}$, confounds the control-flow of all variants, obtaining a single variant that includes all executions occurring in any variant. The information about which transitions are associated with which variants is lost. Each feature expression ψ defined over $\mathbb{F}$ is replaced with *true* if there exists at least one configuration from $\mathbb{K}$ that satisfies ψ. The new abstract set of features is empty: $\boldsymbol{\alpha}^{\text{join}}(\mathbb{F}) = \emptyset$, and the abstract set of valid configurations is a singleton: $\boldsymbol{\alpha}^{\text{join}}(\mathbb{K}) = \{\mathit{true}\}$ if $\mathbb{K} \neq \emptyset$. 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:

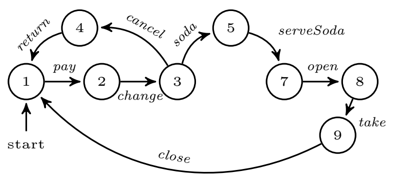
$$\boldsymbol{\alpha}^{\text{join}}(\psi) = \begin{cases} \mathit{true} & \text{if } \exists k \in \mathbb{K}. k \models \psi \\ \mathit{false} & \text{otherwise} \end{cases} \qquad \begin{array}{l} \boldsymbol{\gamma}^{\text{join}}(\mathit{true}) = \mathit{true} \\ \boldsymbol{\gamma}^{\text{join}}(\mathit{false})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

The proposed abstraction-concretization pair is a Galois connection [2] [13,14].

[2] $\langle L, \leq_L \rangle \xleftrightarrow[\alpha]{\gamma} \langle M, \leq_M \rangle$ 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 $l \in L, m \in M$. Here $\leqslant_L$ and $\leqslant_M$ are the pre-order relations for L and M, respectively.



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



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

Fig. 2: Various abstractions of VendingMachine.

The *feature ignore abstraction*, $\boldsymbol{\alpha}_A^{\text{fignore}}$, ignores a single feature $A \in \mathbb{F}$ by confounding the control flow paths that only differ with regard to A, but keeps the precision with respect to control flow paths that do not depend on A. Let ψ be a formula into negation normal form (NNF). We write $\psi[l_A \mapsto \mathit{true}]$ to denote the formula ψ where the literal of A, that is A or $\neg A$, is replaced with *true*. The abstract sets of features and configurations are: $\boldsymbol{\alpha}_A^{\text{fignore}}(\mathbb{F}) = \mathbb{F}\backslash\{A\}$, and $\boldsymbol{\alpha}_A^{\text{fignore}}(\mathbb{K}) = \{k[l_A \mapsto \mathit{true}] \mid k \in \mathbb{K}\}$. The abstraction and concretization functions between $\mathit{FeatExp}(\mathbb{F})$ and $\mathit{FeatExp}(\boldsymbol{\alpha}_A^{\text{fignore}}(\mathbb{F}))$, which form a Galois connection [13,14], are defined as:

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

where ψ and ψ' are in NNF.

The sequential composition $\alpha_2 \circ \alpha_1$ runs two abstractions α_1 and α_2 in sequence (see [13,14] for precise definition). In the following, we will simply write (α, γ) for any Galois connection $\langle \mathit{FeatExp}(\mathbb{F})_{/\equiv}, \models\rangle \overset{\gamma}{\underset{\alpha}{\leftrightarrows}} \langle \mathit{FeatExp}(\alpha(\mathbb{F}))_{/\equiv}, \models\rangle$ constructed using the operators presented in this section.

Given a Galois connection (α, γ) defined on the level of feature expressions, we now induce a notion of abstraction between FTSs.

Definition 5. *Let $\mathcal{F} = (S, \mathit{Act}, \mathit{trans}, I, \mathit{AP}, L, \mathbb{F}, \mathbb{K}, \delta)$ be an FTS, and (α, γ) be a Galois connection. We define $\alpha(\mathcal{F}) = (S, \mathit{Act}, \mathit{trans}, I, \mathit{AP}, L, \alpha(\mathbb{F}), \alpha(\mathbb{K}), \alpha(\delta))$, where $\alpha(\delta) : \mathit{trans} \to \mathit{FeatExp}(\alpha(\mathbb{F}))$ is defined as: $\alpha(\delta)(t) = \alpha(\delta(t))$.*

Example 3. Consider the FTS $\mathcal{F} = \textsc{VendingMachine}$ in Fig. 1a with the set of valid configurations $\mathbb{K} = \{\{v, s\}, \{v, s, t, c, f\}, \{v, s, c\}, \{v, s, c, f\}\}$. We show $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![f]\!]}(\mathcal{F}))$ and $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg f]\!]}(\mathcal{F}))$ in Fig. 2. We do not show transitions labelled with the feature expression *false* and unreachable states. Also note that both $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![f]\!]}(\mathcal{F}))$ and $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg f]\!]}(\mathcal{F}))$ are ordinary TSs, since all transitions are labeled with the feature expression *true*.

For $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![f]\!]}(\mathcal{F}))$ in Fig. 2a, note that $\mathbb{K} \cap [\![f]\!] = \{\{v, s, t, c, f\}, \{v, s, c, f\}\}$. So, transitions annotated with $\neg f$ are not present in $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![f]\!]}(\mathcal{F}))$.

For $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg f]\!]}(\mathcal{F}))$ in Fig. 2b, 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 $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg f]\!]}(\mathcal{F}))$. □

Abstract FTSs have interesting preservation properties [13,14].

Theorem 1 (Soundness). *Let (α, γ) be a Galois connection and $\mathcal{F}$ be an FTS. If $\alpha(\mathcal{F}) \models \phi$, then $\mathcal{F} \models \phi$.*

The family-based model checking problem given in Definition 4 can be reduced to a number of smaller problems by partitioning the set of variants.

Proposition 1. *Let the subsets $\mathbb{K}_1, \mathbb{K}_2, \dots, \mathbb{K}_n$ form a* partition *of the set $\mathbb{K}$. Then: $\mathcal{F} \models \phi$, if and only if, $\pi_{\mathbb{K}_1}(\mathcal{F}) \models \phi, \dots, \pi_{\mathbb{K}_n}(\mathcal{F}) \models \phi$.*

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

In other words, correctness of abstract FTSs implies correctness of the concrete FTS. Note that verification of abstract FTSs can be drastically (even exponentially) faster. However, if abstract FTSs invalidate a property then the concrete FTS may still satisfy the property, i.e. the found counterexample in abstract FTSs may be spurious. In this case, we need to refine the abstract FTSs in order to eliminate the spurious counterexample.

Example 4. Recall the formula $\phi = \Box(\texttt{selected} \implies \Diamond\, \texttt{open})$ 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. 2. First, we can successfully verify that $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg f]\!]}(\textsc{VendingMachine})) \models \phi$, which implies that all valid variants from $\mathbb{K}$ that do not contain the feature f (those are $\{v, s\}$ and $\{v, s, c\}$) satisfy the property ϕ. On the other hand, we have $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![f]\!]}(\textsc{VendingMachine})) \not\models \phi$ with the counterexample: ① $\rightarrow$ ③ $\rightarrow$ ⑤ $\rightarrow$ ⑦ $\rightarrow$ ① $\rightarrow \dots$. This counterexample is genuine for the variants from $\mathbb{K}$ that contain the feature f (those are $\{v, s, t, c, f\}$ and $\{v, s, c, f\}$). In this way, the problem of verifying the FTS $\textsc{VendingMach.}$ against ϕ can be reduced to verifying whether two TSs, $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg f]\!]}(\textsc{VendingMach.}))$ and $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![f]\!]}(\textsc{VendingMach.}))$, satisfy ϕ. $\Box$

3 Abstraction Refinement

We now describe the abstraction refinement procedure (ARP), which uses spurious counterexamples to iteratively refine abstract variational models until either a genuine counterexample is found or the property satisfaction is shown for each variant in the family. Thus, the ARP determines for each variant whether or not it satisfies a property, and provides a counterexample for each variant that do not satisfy the given property.

The ARP for checking $\mathcal{F} \models \phi$, where $\mathcal{F} = (S, Act, trans, I, AP, L, \mathbb{F}, \mathbb{K}, \delta)$, is illustrated in Fig. 3. We apply an initial abstraction α, thus obtaining an initial abstract variational model $\alpha(\mathcal{F})$. If the initial abstract model satisfies the given property, then all variants satisfy it and we stop. Otherwise, the model checker returns a counterexample. Let ψ be the feature expression computed by conjoining feature expressions labelling all transitions that belong to this counterexample in $\mathcal{F}$. There are two cases to consider.

First, if ψ is satisfiable and $\mathbb{K} \cap [\![\psi]\!] \neq \emptyset$, then the found counterexample is *genuine* for variants in $\mathbb{K} \cap [\![\psi]\!]$. For the other variants from $\mathbb{K} \cap [\![\neg\psi]\!]$, the

The ARP checks $\mathcal{F} \models \phi$, where $\mathcal{F}=(S, Act, trans, I, AP, L, \mathbb{F}, \mathbb{K}, \delta)$.

1 Let α be the initial abstraction used to build $\alpha(\mathcal{F})$. Check $\alpha(\mathcal{F}) \models \phi$?

2 If the property is satisfied, then return that ϕ is satisfied for all variants in $\mathbb{K}$.

3 Otherwise, if a genuine (feasible) counterexample is found, let ψ be the feature expression obtained by conjoining the guards $\delta(t)$ over all transitions t appearing in the execution of this counterexample in $\mathcal{F}$. Since the execution is feasible, it follows that ψ is satisfiable. Report that the property is violated for variants in $\mathbb{K} \cap [\![\psi]\!]$. We generate $\mathcal{F}' = \pi_{[\![\neg\psi]\!]}(\mathcal{F})$, and call the ARP to check $\mathcal{F}' \models \phi$ for variants in $\mathbb{K}' = \mathbb{K} \cap [\![\neg\psi]\!]$.

4 Otherwise, if a spurious (infeasible) counterexample is found, let ψ be the feature expression obtained by conjoining the guards $\delta(t)$ over all transitions t appearing in the execution of this counterexample in $\mathcal{F}$. Since the execution is infeasible, it follows that $\psi \wedge (\bigvee_{k\in\mathbb{K}} k)$ is unsatisfiable. Find $\psi' = \texttt{CraigInterpolation}(\psi, \mathbb{K})$. We generate $\mathcal{F}_1 = \pi_{[\![\psi']\!]}(\mathcal{F})$ and $\mathcal{F}_2 = \pi_{[\![\neg\psi']\!]}(\mathcal{F})$, and call the ARP two times to check $\mathcal{F}_1 \models \phi$ for variants in $\mathbb{K}_1 = \mathbb{K} \cap [\![\psi']\!]$ and $\mathcal{F}_2 \models \phi$ for variants in $\mathbb{K}_2 = \mathbb{K} \cap [\![\neg\psi']\!]$. By construction, both $\mathcal{F}_1$ and $\mathcal{F}_2$ do not contain this spurious counterexample.

Fig. 3: The Abstraction Refinement Procedure (ARP)

found counterexample cannot be executed (i.e. the counterexample is spurious for $\mathbb{K} \cap [\![\neg\psi]\!]$). Therefore, we call the ARP again to verify $\pi_{[\![\neg\psi]\!]}(\mathcal{F})$ with updated set of valid configurations $\mathbb{K} \cap [\![\neg\psi]\!]$.

Second, if $\psi \wedge (\bigvee_{k\in\mathbb{K}} k)$ is unsatisfiable (i.e. $\mathbb{K} \cap [\![\psi]\!] = \emptyset$), then the found counterexample is *spurious* for all variants in $\mathbb{K}$ (due to incompatible feature expressions). Now, we describe how a feature expression ψ' used for constructing refined abstract models is determined by means of Craig interpolation [27] from ψ and $\mathbb{K}$. First, we find the minimal unsatisfiable core ψ^c of $\psi \wedge (\bigvee_{k\in\mathbb{K}} k)$, which contains a subset of conjuncts in $\psi \wedge (\bigvee_{k\in\mathbb{K}} k)$, such that ψ^c is still unsatisfiable and if we drop any single conjunct in ψ^c then the result becomes satisfiable. We group conjuncts in ψ^c in two groups X and Y such that $\psi^c = X \wedge Y = false$. Then, the interpolant ψ' is such that: 1) $X \implies \psi'$, 2) $\psi' \wedge Y = false$, 3) ψ' refers only to common variables of X and Y. Intuitively, we can think of the interpolant ψ' as a way of filtering out irrelevant information from X. In particular, ψ' summarizes and translates why X is inconsistent with Y in their shared language. Once the interpolant ψ' is computed, we call the ARP to check $\pi_{[\![\psi']\!]}(\mathcal{F}) \models \phi$ for variants in $\mathbb{K} \cap [\![\psi']\!]$, and $\pi_{[\![\neg\psi']\!]}(\mathcal{F}) \models \phi$ for variants in $\mathbb{K} \cap [\![\neg\psi']\!]$. By construction, we guarantee that the found spurious counterexample does not occur neither in $\pi_{[\![\psi']\!]}(\mathcal{F})$ nor in $\pi_{[\![\neg\psi']\!]}(\mathcal{F})$.

Note that, in Step **1**, the initial abstraction can be chosen arbitrarily. This choice does not affect correctness and termination of the ARP, but it allows experimentation with different heuristics in concrete implementations. For example, if we use the initial abstraction $\boldsymbol{\alpha}^{\text{join}}$, then as an abstract model we obtain an ordinary TS where all feature expressions associated with transitions of $\mathcal{F}$ occurring in some valid variant are replaced with *true*. Therefore, the verification step can be performed using a single-system model checker (e.g. SPIN). Also note that we call the ARP until there are no more counterexamples or the updated set of valid configurations $\mathbb{K}$ becomes empty.

Example 5. Let $\mathcal{F}$ be VendingMachine of Fig. 1a with configurations $\mathbb{K} = \{\{v,s\},\{v,s,t,c,f\},\{v,s,c\},\{v,s,c,f\}\}$. Let $\boldsymbol{\alpha}^{\text{join}}$ be the initial abstraction.

We check $\mathcal{F} \models \phi$, where $\phi = \Box(\texttt{selected} \implies \Diamond\,\texttt{open})$ using the ARP. We first check $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F}) \models \phi$? The following spurious counterexample is reported: $① \xrightarrow{pay} ② \xrightarrow{change} ③ \xrightarrow{tea} ⑥ \xrightarrow{serveTea} ⑦ \xrightarrow{take} ① \ldots$. The associated feature expression in $\mathcal{F}$ is: $(v \land \neg f) \land v \land t \land f$. The minimal unsatisfiable core is: $(v \land \neg f) \land f$, 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 counterexample. Thus, in the next iteration we check $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg f]\!]}(\mathcal{F})) \models \phi$ and $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![f]\!]}(\mathcal{F})) \models \phi$, which give conclusive results for all variants from $\mathbb{K}$ as explained in Example 4.

Consider the property $\phi' = \Box\Diamond\,\texttt{open}$. The following counterexample is found in $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F})$: $① \xrightarrow{pay} ② \xrightarrow{change} ③ \xrightarrow{cancel} ④ \xrightarrow{return} ① \ldots$. The associated feature expression in $\mathcal{F}$ is: $v \land \neg f \land c$, so this is a genuine counterexample for the variant $\{v,s,c\} \in \mathbb{K}$. In the next iteration, we check $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg(v\land\neg f\land c)]\!]}(\mathcal{F})) \models \phi'$ for variants $\mathbb{K}\backslash\{v,s,c\}$. We obtain the counterexample: $① \xrightarrow{free} ③ \xrightarrow{cancel} ④ \xrightarrow{return} ① \ldots$, with associated feature expression $f \land c$, realizable for variants $\{v,s,t,c,f\}$ and $\{v,s,c,f\}$). In the final iteration, we check $\boldsymbol{\alpha}^{\text{join}}\big(\pi_{[\![\neg(f\land c)]\!]}(\pi_{[\![\neg(v\land\neg f\land c)]\!]}(\mathcal{F}))\big) \models \phi'$ for the variant $\{v,s\}$. The property holds, so ϕ' is satisfied by $\{v,s\}$. $\Box$

Theorem 2. *The ARP terminates and is correct.*

Proof. At the end of an iteration, the ARP either terminates with answer 'yes', or finds a genuine counterexample and updates $\mathbb{K}$ into $\mathbb{K}'$, or finds a spurious counterexample and updates $\mathbb{K}$ into $\mathbb{K}_1$ and $\mathbb{K}_2$. Given that $\mathbb{K}' \subset \mathbb{K}$ (the counterexample is genuine for some non-empty subset of $\mathbb{K}$), and $\mathbb{K}_1 \subset \mathbb{K}$, $\mathbb{K}_2 \subset \mathbb{K}$ (by def. $\mathbb{K}_1 \neq \emptyset$, $\mathbb{K}_2 \neq \emptyset$, $\mathbb{K}_1 \cup \mathbb{K}_2 = \mathbb{K}$), the number of possible updates and calls to the ARP are finite. Therefore, the number of iterations is also finite.

If the ARP terminates with answer that a property is satisfied (resp., property is not satisfied) by a variant, then the answer is correct by Theorem 1, since any abstraction constructs an over-approximated model for a given set of variants. $\Box$

4 Evaluation

In this section, we describe our implementation of the ARP, and present the results of experiments carried out on several variational models. We use experiments to evaluate in which cases and to what extent our ARP technique outperforms the family-based model checking algorithms of FTS [7,8] implemented in $\overline{\text{SNIP}}$ [3].

[3] The project on development of the $\overline{\text{SNIP}}$ tool (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.

Implementation. It is difficult to use FTSs directly to model large variational systems. Therefore, $\overline{\text{SNIP}}$ uses the high-level languages *f*PROMELA and *TVL* for modeling variational systems and their configuration sets, respectively. *f*PROMELA is an extension of PROMELA, the language of the SPIN model checker [19], adding *feature variables*, $\mathbb{F}$, and a new *guarded-by-features statement*, "`gd`". The "`gd`" is a non-deterministic statement similar to PROMELA's "`if`", except that only feature expressions can be used as guards. Actually, this is the only place where features may be used. Thus, "`gd`" plays the same role in *f*PROMELA as "`#ifdef`" in the C Preprocessor [24]. *TVL* [6] is a textual modelling language for describing the set of valid configurations, $\mathbb{K}$, for an *f*PROMELA model along with all available features, $\mathbb{F}$. It has been shown in [13,14] that variability abstractions and projections can be implemented as syntactic source-to-source transformations of *f*PROMELA and *TVL* models, which enable an effective computation of abstract models syntactically from high-level modelling languages. More precisely, let M and T be *f*PROMELA and *TVL* models, and let $[\![M]\!]_T$ represent the FTS obtained by their compilation. Since variability abstractions affect only variability-specific aspects of a system, for any abstraction α we can define $\alpha(M)$ and $\alpha(T)$ as syntactic transformations such that $\alpha([\![M]\!]_T) = [\![\alpha(M)]\!]_{\alpha(T)}$. That is, the abstract model obtained by applying α on the FTS $[\![M]\!]_T$ coincides with the FTS obtained by compiling $\alpha(M)$ and $\alpha(T)$. The same applies for projections $\pi_{[\![\psi]\!]}$. The *f*PROMELA RECONFIGURATOR tool [13,14] syntactically calculates the transformations corresponding to abstractions and projections. This is important for two reasons. First, it allows to easily implement our technique based on abstractions and projections. Second, we avoid the need for intermediate storage in memory of the concrete full-blown FTSs. In our implementation of the ARP, we use $\boldsymbol{\alpha}^{\text{join}}$ as the initial abstraction. Hence, after applying $\boldsymbol{\alpha}^{\text{join}}$ on *f*PROMELA and *TVL* models M and T, we obtain an ordinary PROMELA model and we call SPIN to check $[\![\alpha(M)]\!]_{\alpha(T)} \models \phi$? If a counterexample trace is returned, we inspect the error trace in detail by using SPIN's simulation mode. We replay the error trace through $\alpha(M)$ and M simultaneously, and we find the feature expression ψ that characterizes this trace in M. In order to do this, we use the fact that $\alpha(M)$ and M have the same control structures (same number of lines and statements), except that "`gd`" statements in M are replaced with "`if`" statements in $\alpha(M)$ by the corresponding transformations that affect only their guards.

Experimental setup. For our experiments, we use: a warm-up example to demonstrate specific characteristics of our ARP, and the MINEPUMP [25] variational system whose *f*PROMELA model was created as part of the $\overline{\text{SNIP}}$ project. We verify a range of properties by using (1) the ARP with $\boldsymbol{\alpha}^{\text{join}}$ as the initial abstraction and SPIN as the verification tool (denoted ARP+SPIN), and by using (2) plain family-based model checking with $\overline{\text{SNIP}}$. The reported performance numbers constitute the average runtime of five independent executions. For each experiment, we measure: TIME which is the time to verify 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). For the ARP, along with the total time the ARP takes to complete we also report in parentheses the time taken by

```
typedef features {
  bool A1; bool A2; ...bool An}
features f
active proctype foo() {
  int i = 0;
  gd :: f.A1 ⇒ i++ :: else ⇒ skip dg;
  gd :: f.A2 ⇒ i++ :: else ⇒ skip dg;
      ......
  gd :: f.An ⇒ i++ :: else ⇒ skip dg;
  assert(i ≥ k) }
```

(a) An fPromela model $\mathcal{F}$.

```
typedef features {
  bool A1; bool A2; ...bool An}
features f
active proctype foo() {
  int i = 0;
  if :: true ⇒ i++ :: true ⇒ skip fi;
  if :: true ⇒ i++ :: true ⇒ skip fi;
      ......
  if :: true ⇒ i++ :: true ⇒ skip fi;
  assert(i ≥ k) }
```

(b) A Promela model $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F})$

Fig. 4: An fPromela model and the corresponding $\boldsymbol{\alpha}^{\text{join}}$ abstract model.

SPIN to perform the actual model checking tasks. The rest of the total time the ARP uses to calculate abstractions, projections, analyze error traces, etc. We only measure the times to generate a process analyser (pan) for SPIN 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. All experiments were executed on a LUbunutuVM 64-bit Intel®Core^{TM} i7-4600U CPU running at 2.10 GHz with 4 GB memory. The implementation, benchmarks, and all results obtained from our experiments are available from: `http://www.itu.dk/people/adim/arp.html`.

Warm-up example. Consider the fPromela model $\mathcal{F}$ given in Fig 4a. After declaring feature variables, `A1`...`An`, the process `foo()` is defined. The first `gd` statement specifies that `i++` is available for variants that contain the feature `A1`, and `skip` for variants with $\neg$`A1`. The following `gd` statements are similar, except that their guards are the features from `A2` to `An`. We want to check the assertion, `i` $\geq k$, where k is a meta-variable that can be replaced with different values: 0, 1, ..., `n`. The corresponding *TVL* model specifies that all features are optional and unconstrained, which means that all possible $2^{\mathtt{n}}$ configurations are valid. We use two approaches to check the above assertions: ARP+SPIN and the family-based model checker $\overline{\text{SNIP}}$. The initial abstract model $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F})$ used in the ARP is shown in Fig 4b. Since there are valid variants where `Aj` is enabled and valid variants where `Aj` is disabled (for any $j \in \{1, \ldots, \mathtt{n}\}$), we have that both statements `i++` and `skip` become available in $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F})$ for all "`gd`" statements.

When $k = 0$, the assertion `i` ≥ 0 is satisfied by all variants. The ARP terminates in one iteration with only one call to SPIN, which reports that $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F})$ satisfies the assertion. When $k = 1$, the ARP needs two iterations to find a (genuine) counterexample which corresponds to a single configuration where all features are disabled, and to certify that all other variants satisfy the assertion. When $k = 2$, the ARP runs in $\mathtt{n} + 1$ iterations producing $\mathtt{n} + 1$ erroneous variants: one variant where all features are disabled, and `n` variants where exactly one feature is enabled and all others are disabled. When $k = \mathtt{n}$, the ARP will need $\mathtt{n} + 1$ iterations to terminate reporting that there is only one variant, where all features are enabled, that satisfies the assertion `i` $\geq$ `n`. All

n	$k=0$				$k=1$			
	ARP + SPIN		$\overline{\text{SNIP}}$		ARP+SPIN		$\overline{\text{SNIP}}$	
	Time	Space	Time	Space	Time	Space	Time	Space
2	0.35 (0.02)	13	0.34	14	1.16 (0.15)	26	0.34	13
5	0.63 (0.06)	52	0.35	126	2.26 (0.26)	245	0.35	125
10	0.68 (0.06)	177	0.61	4,094	4.29 (0.47)	1,690	0.61	4,093
24	0.71 (0.07)	926	1262.7	67,108,862	8.63 (0.86)	21,696	1318.1	67,108,861
25	0.75 (0.07)	1002	– infeasible –		8.81 (0.88)	24,475	– infeasible –	
100	0.77 (0.08)	15,252	– infeasible –		39.54 (3.81)	1,515,400	– infeasible –	

Fig. 5: Verification of the warm-up example. Time in seconds.

other variants are erroneous. This represents the worst case for our ARP, since all possible variants will be generated explicitly and checked by SPIN in a brute-force fashion. In addition, we have the overhead of generating all intermediate projections and abstractions as well as their verification with SPIN, for which spurious counterexamples are obtained. The performance results are shown in Fig. 5. We say that a task is *infeasible* when it is taking more time than the given timeout threshold, which we set on 1 hour. Notice that $\overline{\text{SNIP}}$ reports the correct results in only one iteration for all cases. Yet, as shown in Fig. 5, for `n` $= 25$ (for which $|\mathbb{K}| = 2^{25} = 33,554,432$ variants) $\overline{\text{SNIP}}$ timeouts after visiting 150 M states. On the other hand, our ARP based approach is feasible even for very large values of `n` when k is smaller (see Fig. 5). In general, the ARP aims to partition the configuration space into subspaces that satisfy and violate the property at hand. When k is higher, that split becomes more irregular and the ARP needs to perform more iterations and calls to SPIN to find it automatically. Therefore, in those cases it takes more time to complete.

MinePump. The MinePump variational system is given by an *f*Promela model with 200 LOC and a *TVL* model that contains 7 independent optional features: `Start`, `Stop`, `MethaneAlarm`, `MethaneQuery`, `Low`, `Normal`, and `High`, thus yielding $2^7 = 128$ variants. The FTS of MinePump has 21,177 states. It consists of 5 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 in the mine is high, but only if there is no methane within it.

For evaluation, we consider five interesting properties of MinePump (taken from [8]). First, we consider three properties, φ_1, φ_2 and φ_3, that are intended to be satisfied by all variants. The property φ_1 is the absence of deadlock; the property φ_2 is that under a *fairness assumption* (the system will infinity often read messages of various types) the pump is never indefinitely off when the water level is high and there is no methane; whereas the property φ_3 is that if the pump is switched on then the controller state is running. For all three properties, the ARP terminates after one iteration reporting that the properties are satisfied by all variants. Then, we have two properties, φ_4 and φ_5, which are

prop- *-erty*	ARP + SPIN		$\overline{\text{SNIP}}$	
	TIME	SPACE	TIME	SPACE
φ_1	0.79 (0.11) s	36,725	1.96 s	250,770
φ_2	0.91 (0.21) s	266,601	3.76 s	441,063
φ_3	0.85 (0.12) s	36,725	2.64 s	326,064
φ_4	8.15 (1.91) s	57,065	8.95 s	398,167
φ_5	7.82 (1.89) s	32,532	7.89 s	218,552

Fig. 6: Verification of MINEPUMP properties.

satisfied by some variants and violated by others, such that there are different counterexamples corresponding to violating variants. The property φ_4 (when the water is high and there is no methane, the pump will not be switched on at all eventually) is violated by variants that satisfy `Start` $\land$ `High` (32 variants in total). The property φ_5 (when the water is low, then the pump will be off) is also violated by variants satisfying `Start` $\land$ `High`. For both properties, our ARP runs in seven iterations, producing 12 different counterexamples for φ_4 and 13 different counterexamples for φ_5. Fig. 6 shows the performance results of verifying properties, φ_1 to φ_5, using our ARP with SPIN approach and the $\overline{\text{SNIP}}$. The ARP achieves improvements in both TIME and SPACE in most cases, especially for properties φ_1 to φ_3 satisfied by all variants which are verified in only one iteration. Of course, the performances of the ARP will start to decline for properties for which the ARP needs higher number of iterations and calls to SPIN in order to complete. However, we can see that for both φ_4 and φ_5 the actual verification time taken by SPIN (given in parentheses) in our ARP is still considerable smaller than the time taken by $\overline{\text{SNIP}}$. Still, in these cases we obtain very long counterexamples (around thousand steps) so the ARP will need some additional time to process them.

Discussion. In conclusion, the ARP achieves the best results when the property to be checked is either satisfied by all variants or only a few erroneous variants exist. In those cases, the ARP will report conclusive results in few iterations. The worst case is when every variant triggers a different counterexample, so our ARP ends up in verifying all variants one by one in a brute-force fashion (plus the overhead for generating and verifying all intermediate abstract models). Variability abstractions weaken feature expressions used in FTSs, thus increasing the commonality between the behaviours of variants. In the case of $\boldsymbol{\alpha}^{\text{join}}$ this enables the use of (single-system) SPIN model checker. SPIN is a highly-optimized industrial-strength tool which is much faster than 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. Note that we can also implement the ARP to work with $\overline{\text{SNIP}}$ by using $\boldsymbol{\alpha}^{\text{fignore}}$ instead of $\boldsymbol{\alpha}^{\text{join}}$ as the initial abstraction. The ARP will work correctly for any choice of features to be ignored by $\boldsymbol{\alpha}^{\text{fignore}}$. However, in order the ARP to terminate faster and achieve some speedups, the ignored features should be chosen carefully by exploiting the knowledge of the variational system and property at hand.

5 Related Work

Family-based (lifted) analyses and verification techniques have been a topic of considerable research recently (see [30] for a survey). Some successful examples are lifted syntax checking [24,17], lifted type checking [23], lifted static data-flow analysis [3,28,15,16], lifted verification [21,29,20], etc.

In the context of family-based model checking, one of the earliest attempts for modelling variational systems is by using modal transition systems (MTSs) [26,2]. Following this, Classen et al. present FTSs in [7,8] and show how specifically designed family-based model checking algorithms (implemented in $\overline{\text{SNIP}}$) can be used for verifying FTSs against fLTL properties. An FTS-specific verification procedure based on counterexample guided abstraction refinement has been proposed in [10]. Abstractions on FTSs are introduced by using existential F-abstraction functions (as opposed to Galois connections here), and simulation relation is used to relate different abstraction levels. There are other important differences between the approach in [10] and our ARP. Refinement of feature abstractions in [10] is defined by simply replacing the abstract (weakened) feature expressions occurring in transitions of the spurious counterexample by their concrete feature expressions. In contrast, we use Craig interpolation as well as suitable combinations of variability abstractions and projections to generate refined abstract models. The abstractions in [10] are applied on feature program graphs (an intermediate structure between high-level fPROMELA models and FTSs) in $\overline{\text{SNIP}}$. In contrast, we apply variability abstractions as preprocessor transformations directly on high-level fPROMELA models thus avoiding to generate any intermediate concrete semantic model in the memory. In the case of $\boldsymbol{\alpha}^{\text{join}}$, this leads to generating PROMELA models and using SPIN for the ARP. The work [12] presents an approach for family-based software model checking of `#ifdef`-based second-order program families using symbolic game semantics models [11].

6 Conclusion

In this work we have proposed an automatic abstraction refinement procedure for family-based model checking of variational systems. Automatic refinement gives us an adaptive divide-and-conquer strategy for the configuration space. The obtained tool represents a completely automatic alternative to the family-based model checker $\overline{\text{SNIP}}$, which is simpler, easier to maintain, and more efficient for some interesting properties than $\overline{\text{SNIP}}$. It automatically benefits from all optimizations of SPIN. The overall design principle is general and can be applied to lifting of other automatic verification tools to variational systems.

References

1. Baier, C., Katoen, J.: Principles of model checking. MIT Press (2008)
2. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: 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), http://dx.doi.org/10.1016/j.jlamp.2015.09.004
3. Bodden, E., Tolêdo, T., Ribeiro, M., Brabrand, C., Borba, P., Mezini, M.: Spl $^{\text{lift}}$: statically analyzing software product lines in minutes instead of years. In: ACM SIGPLAN Conference on PLDI '13. pp. 355–364 (2013)
4. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Computer Aided Verification, 12th International Conference, CAV 2000, Proceedings. Lecture Notes in Computer Science, vol. 1855, pp. 154–169. Springer (2000), http://dx.doi.org/10.1007/10722167_15
5. Clarke, E.M., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI-C programs using SAT. Formal Methods in System Design 25(2-3), 105–127 (2004), http://dx.doi.org/10.1023/B:FORM.0000040025.89719.f3
6. Classen, A., Boucher, Q., Heymans, P.: A text-based approach to feature modelling: Syntax and semantics of TVL. Sci. Comput. Program. 76(12), 1130–1143 (2011), http://dx.doi.org/10.1016/j.scico.2010.10.005
7. Classen, A., Cordy, M., Heymans, P., Legay, A., Schobbens, P.: Model checking software product lines with SNIP. STTT 14(5), 589–612 (2012), http://dx.doi.org/10.1007/s10009-012-0234-1
8. Classen, A., Cordy, M., Schobbens, P., Heymans, P., Legay, A., Raskin, J.: 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), http://doi.ieeecomputersociety.org/10.1109/TSE.2012.86
9. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley (2001)
10. Cordy, M., Heymans, P., Legay, A., Schobbens, P., Dawagne, B., Leucker, M.: Counterexample guided abstraction refinement of product-line behavioural models. In: Cheung, S., Orso, A., Storey, M.D. (eds.) Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22). pp. 190–201. ACM (2014), http://doi.acm.org/10.1145/2635868.2635919
11. Dimovski, A.S.: Program verification using symbolic game semantics. Theor. Comput. Sci. 560, 364–379 (2014), http://dx.doi.org/10.1016/j.tcs.2014.01.016
12. Dimovski, A.S.: Symbolic game semantics for model checking program families. In: Model Checking Software - 23nd International Symposium, SPIN 2016, Proceedings. LNCS, vol. 9641, pp. 19–37. Springer (2016)
13. Dimovski, A.S., Al-Sibahi, A.S., Brabrand, C., Wasowski, A.: Family-based model checking without a family-based model checker. In: Model Checking Software - 22nd International Symposium, SPIN 2015, Proceedings. LNCS, vol. 9232, pp. 282–299. Springer (2015), http://dx.doi.org/10.1007/978-3-319-23404-5_18
14. Dimovski, A.S., Al-Sibahi, A.S., Brabrand, C., Wasowski, A.: Efficient family-based model checking via variability abstractions. STTT (2016)
15. Dimovski, A.S., Brabrand, C., Wasowski, A.: Variability abstractions: Trading precision for speed in family-based analyses. In: 29th European Conference on Object-Oriented Programming, ECOOP 2015. LIPIcs, vol. 37, pp. 247–270. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015), http://dx.doi.org/10.4230/LIPIcs.ECOOP.2015.247

16. Dimovski, A.S., Brabrand, C., Wasowski, A.: Finding suitable variability abstractions for family-based analysis. In: FM 2016: Formal Methods - 21st International Symposium, Proceedings. LNCS, vol. 9995, pp. 217–234 (2016), http://dx.doi.org/10.1007/978-3-319-48989-6_14
17. Gazzillo, P., Grimm, R.: Superc: parsing all of C by taming the preprocessor. In: Vitek, J., Lin, H., Tip, F. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012. pp. 323–334. ACM (2012), http://doi.acm.org/10.1145/2254064.2254103
18. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004. pp. 232–244. ACM (2004), http://doi.acm.org/10.1145/964001.964021
19. Holzmann, G.J.: The SPIN Model Checker - primer and reference manual. Addison-Wesley (2004)
20. Iosif-Lazar, A.F., Al-Sibahi, A.S., Dimovski, A.S., Savolainen, J.E., Sierszecki, K., Wasowski, A.: Experiences from designing and validating a software modernization transformation (E). In: 30th IEEE/ACM Int. Conf. on Automated Software Engineering, ASE 2015. pp. 597–607 (2015), http://dx.doi.org/10.1109/ASE.2015.84
21. Iosif-Lazar, A.F., Melo, J., Dimovski, A.S., Brabrand, C., Wasowski, A.: Effective analysis of c programs by rewriting variability. The Art, Science, and Engineering of Programming, Programming'17 (2017)
22. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) feasibility study. Tech. rep., Carnegie-Mellon University Software Engineering Institute (November 1990)
23. Kästner, C., Apel, S., Thüm, T., Saake, G.: Type checking annotation-based product lines. ACM Trans. Softw. Eng. Methodol. 21(3), 14 (2012)
24. Kästner, C., Giarrusso, P.G., Rendel, T., Erdweg, S., Ostermann, K., Berger, T.: Variability-aware parsing in the presence of lexical macros and conditional compilation. In: Proceedings of the 26th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011. pp. 805–824 (2011), http://doi.acm.org/10.1145/2048066.2048128
25. Kramer, J., Magee, J., Sloman, M., Lister, A.: Conic: An integrated approach to distributed computer control systems. IEE Proc. 130(1), 1–10 (1983)
26. Larsen, K.G., Nyman, U., Wasowski, A.: Modal I/O automata for interface and product line theories. In: Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Proceedings. LNCS, vol. 4421, pp. 64–79. Springer (2007), http://dx.doi.org/10.1007/978-3-540-71316-6_6
27. McMillan, K.L.: Applications of craig interpolants in model checking. In: Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Proceedings. Lecture Notes in Computer Science, vol. 3440, pp. 1–12. Springer (2005), http://dx.doi.org/10.1007/978-3-540-31980-1_1
28. Midtgaard, J., Dimovski, A.S., Brabrand, C., Wasowski, A.: Systematic derivation of correct variability-aware program analyses. Sci. Comput. Program. 105, 145–170 (2015), http://dx.doi.org/10.1016/j.scico.2015.04.005
29. von Rhein, A., Thüm, T., Schaefer, I., Liebig, J., Apel, S.: Variability encoding: From compile-time to load-time variability. J. Log. Algebr. Meth. Program. 85(1), 125–145 (2016), http://dx.doi.org/10.1016/j.jlamp.2015.06.007
30. Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G.: A classification and survey of analysis strategies for software product lines. ACM Comput. Surv. 47(1), 6 (2014), http://doi.acm.org/10.1145/2580950