Abstract Family-based Model Checking using Modal Featured Transition Systems: Preservation of CTL* (Extended Version)

Aleksandar S. Dimovski

Faculty of Informatics, Mother Teresa University, Skopje, Mkd

Abstract. Variational systems allow effective building of many custom variants by using features (configuration options) to mark the variable functionality. In many of the applications, their quality assurance and formal verification are of paramount importance. Family-based model checking allows simultaneous verification of all variants of a variational system in a single run by exploiting the commonalities between the variants. Yet, its computational cost still greatly depends on the number of variants (often huge).

In this work, we show how to achieve efficient family-based model checking of CTL* temporal properties using variability abstractions and off-the-shelf (single-system) tools. We use variability abstractions for deriving abstract family-based model checking, where the variability model of a variational system is replaced with an abstract (smaller) version of it, called *modal featured transition system*, which preserves the satisfaction of both universal and existential temporal properties, as expressible in CTL*. Modal featured transition systems contain two kinds of transitions, termed may and must transitions, which are defined by the conservative (over-approximating) abstractions and their dual (under-approximating) abstractions, respectively. The variability abstractions can be combined with different partitionings of the set of variants to infer suitable divide-and-conquer verification plans for the variational system. We illustrate the practicality of this approach for several variational systems.

1 Introduction

Variational systems appear in many application areas and for many reasons. Efficient methods to achieve customization, such as *Software Product Line Engineering* (SPLE) [10], use *features* (configuration options) to control presence and absence of the variable functionality [1]. Family members, called *variants* of a *variational system*, are specified in terms of features selected for that particular variant. The reuse of code common to multiple variants is maximized. The SPLE method is particularly popular in the embedded and critical system domain (e.g. cars, phones). In these domains, a rigorous verification and analysis is very important. Among the methods included in current practices, *model checking* [2] is a well-studied technique used to establish that temporal logic properties hold for a system.

Variability and SPLE are major enablers, but also a source of complexity. Obviously, the size of the configuration space (number of variants) is the limiting factor to the feasibility of any verification technique. Exponentially many variants can be derived from few configuration options. This problem is referred to as *the configuration space explosion* problem. A simple "brute-force" application of a single-system model checker to each variant is infeasible for realistic variational systems, due to the sheer number of variants. This is very ineffective also because the same execution behavior is checked multiple times, whenever it is shared by some variants. Another, more efficient, verification technique [8,7] is based on using compact representations for modelling variational systems, which incorporate the commonality within the family. We will call these representations variability models (or featured transition systems). Each behavior in a variability model is associated with the set of variants able to produce it. A specialized family-based model checking algorithm executed on such a model, checks an execution behavior only once regardless of how many variants include it. These algorithms model check all variants simultaneously in a single run and pinpoint the variants that violate properties. Unfortunately, their performance *still* heavily depends on the size and complexity of the configuration space of the analyzed variational system. Moreover, maintaining specialized family-based tools is also an expensive task.

In order to address these challenges, we propose to use standard, single-system model checkers with an alternative, externalized way to combat the configuration space explosion. We apply the so-called *variability abstractions* to a variability model which is too large to handle ("configuration space explosion"), producing a more *abstract model*, which is smaller than the original one. We abstract from certain aspects of the configuration space, so that many of the configurations (variants) become indistinguishable and can be collapsed into a single abstract configuration. The abstract model is constructed in such a way that if some property holds for this abstract model it will also hold for the concrete model. Our technique extends the scope of existing over-approximating variability abstractions [16,21] which currently support the verification of universal properties only (LTL and $\forall$CTL). Here we construct abstract variability models which can be used to check arbitrary formulae of CTL*, thus including arbitrary nested path quantifiers. We use modal featured transition systems (MFTSs) for representing abstract variability models. MFTSs are featured transition systems (FTSs) with two kinds of transitions, *must* and *may*, expressing behaviours that necessarily occur (must) or possibly occur (may). We use the standard conservative (over-approximating) abstractions to define may transitions, and their dual (under-approximating) abstractions to define must transitions. Therefore, MFTSs perform both over- and under-approximation, admitting both universal and existential properties to be deduced. Since MFTSs preserve all CTL* properties, we can verify any such properties on the concrete variability model (which is given as an FTSs) by verifying these on an abstract MFTS. Any model checking problem on modal transitions systems (resp., MFTSs) can be reduced to two traditional model checking problems on standard transition systems (resp., FTSs). The overall technique relies on partitioning and abstracting concrete FTSs, until the point we obtain models with so limited variability (or, no variability) that it is feasible to complete their model checking in the brute-force fashion using the standard single-system model checkers. Compared to the family-based model checking, experiments show that the proposed technique achieves significant performance gains .

2 Background

In this section, we present the background used in later developments.

Modal 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. We assume that only a subset $\mathbb{K} \subseteq 2^{\mathbb{F}}$ of configurations are *valid*. 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.

We recall the basic definition of a transition system (TS) and a modal transition system (MTS) that we will use to describe behaviors of single-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; $I \subseteq S$ is a set of initial states; AP is a set of atomic propositions; and $L : S \to 2^{AP}$ is a labelling function specifying which propositions hold in a state. We write $s_1 \xrightarrow{\lambda} s_2$ whenever $(s_1, \lambda, s_2) \in trans$.*

An *execution* (behaviour) of a TS $\mathcal{T}$ is an *infinite* sequence $\rho = s_0 \lambda_1 s_1 \lambda_2 \ldots$ 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.

MTSs [29] are a generalization of transition systems that allows describing not just a sum of all behaviors of a system but also an over- and under-approximation of the system's behaviors. An MTS is a TS equipped with two transition relations: *must* and *may*. The former (must) is used to specify the required behavior, while the latter (may) to specify the allowed behavior of a system.

Definition 2. *A modal transition system (MTS) is represented by a tuple $\mathcal{M} = (S, Act, trans^{may}, trans^{must}, I, AP, L)$, where $trans^{may} \subseteq S \times Act \times S$ describe may transitions of $\mathcal{M}$; $trans^{must} \subseteq S \times Act \times S$ describe must transitions of $\mathcal{M}$, such that $trans^{must} \subseteq trans^{may}$.*

The intuition behind the inclusion $trans^{must} \subseteq trans^{may}$ is that transitions that are necessarily true ($trans^{must}$) are also possibly true ($trans^{may}$). A *may-execution* in $\mathcal{M}$ is an execution with all its transitions in $trans^{may}$; whereas a *must-execution* in $\mathcal{M}$ is an execution with all its transitions in $trans^{must}$. We use $[\![\mathcal{M}]\!]^{may}_{MTS}$ to denote the set of all may-executions in $\mathcal{M}$, whereas $[\![\mathcal{M}]\!]^{must}_{MTS}$ to denote the set of all must-executions in $\mathcal{M}$.

An FTS describes behavior of a whole family of systems in a *superimposed* manner. This means that it combines models of many variants in a single monolithic description, where the transitions are guarded by a *presence condition* that identifies the variants they belong to. The presence conditions ψ are drawn from the set of feature expressions, $\mathit{FeatExp}(\mathbb{F})$, which are propositional logic formulae over $\mathbb{F}$: $\psi ::= \mathit{true} \mid A \in \mathbb{F} \mid \neg\psi \mid \psi_1 \wedge \psi_2$. The presence condition ψ of a transition specifies the variants in which the transition is enabled. We write $[\![\psi]\!]$ to denote the set of variants from $\mathbb{K}$ that satisfy ψ, i.e. $k \in [\![\psi]\!]$ iff $k \models \psi$.

Definition 3. *A featured transition system (FTS) represents a tuple* $\mathcal{F} = (S, \mathit{Act}, \mathit{trans}, I, \mathit{AP}, L, \mathbb{F}, \mathbb{K}, \delta)$*, where* $S, \mathit{Act}, \mathit{trans}, I, \mathit{AP}$*, and* L *are defined as in TS;* $\mathbb{F}$ *is the set of available features;* $\mathbb{K}$ *is a set of valid configurations; and* $\delta : \mathit{trans} \to \mathit{FeatExp}(\mathbb{F})$ *is a total function decorating transitions with presence conditions (feature expressions).*

The *projection* of an FTS $\mathcal{F}$ to a variant $k \in \mathbb{K}$, denoted as $\pi_k(\mathcal{F})$, is the TS $(S, \mathit{Act}, \mathit{trans}', I, \mathit{AP}, L)$, where $\mathit{trans}' = \{t \in \mathit{trans} \mid k \models \delta(t)\}$. We lift the definition of *projection* to sets of configurations $\mathbb{K}' \subseteq \mathbb{K}$, denoted as $\pi_{\mathbb{K}'}(\mathcal{F})$, by keeping the transitions admitted by at least one of the configurations in $\mathbb{K}'$. That is, $\pi_{\mathbb{K}'}(\mathcal{F})$, is the FTS $(S, \mathit{Act}, \mathit{trans}', I, \mathit{AP}, L, \mathbb{F}, \mathbb{K}', \delta)$, where $\mathit{trans}' = \{t \in \mathit{trans} \mid \exists k \in \mathbb{K}'.k \models \delta(t)\}$. The *semantics* of an FTS $\mathcal{F}$, denoted as $[\![\mathcal{F}]\!]_{FTS}$, is the union of behaviours of the projections on all valid variants $k \in \mathbb{K}$, i.e. $[\![\mathcal{F}]\!]_{FTS} = \cup_{k\in\mathbb{K}}[\![\pi_k(\mathcal{F})]\!]_{TS}$.

We will use modal featured transition systems (MFTS) for representing abstractions of FTSs. MFTSs are variability-aware extension of MTSs.

Definition 4. *A modal featured transition system (MFTS) represents a tuple* $\mathcal{MF} = (S, \mathit{Act}, \mathit{trans}^{may}, \mathit{trans}^{must}, I, \mathit{AP}, L, \mathbb{F}, \mathbb{K}, \delta^{may}, \delta^{must})$*, where* trans^{may} *and* $\delta^{may} : \mathit{trans}^{may} \to \mathit{FeatExp}(\mathbb{F})$ *describe may transitions of* $\mathcal{MF}$*;* trans^{must} *and* $\delta^{must} : \mathit{trans}^{must} \to \mathit{FeatExp}(\mathbb{F})$ *describe must transitions of* $\mathcal{MF}$*.*

The *projection* of an MFTS $\mathcal{MF}$ to a variant $k \in \mathbb{K}$, denoted as $\pi_k(\mathcal{MF})$, is the MTS $(S, \mathit{Act}, \mathit{trans}'^{may}, \mathit{trans}'^{must}, I, \mathit{AP}, L)$, where $\mathit{trans}'^{may} = \{t \in \mathit{trans}^{may} \mid k \models \delta^{may}(t)\}$, $\mathit{trans}'^{must} = \{t \in \mathit{trans}^{must} \mid k \models \delta^{must}(t)\}$. We define $[\![\mathcal{MF}]\!]^{may}_{MFTS} = \cup_{k\in\mathbb{K}}[\![\pi_k(\mathcal{MF})]\!]^{may}_{MTS}$, and $[\![\mathcal{MF}]\!]^{must}_{MFTS} = \cup_{k\in\mathbb{K}}[\![\pi_k(\mathcal{MF})]\!]^{must}_{MTS}$.

Example 1. Throughout this paper, we will use a beverage vending machine as a running example [8]. Figure 1 shows the FTS of a VendingMachine family. It has five features, and each of them is assigned an identifying letter and a color. The features are: `VendingMachine` (denoted by letter v, in black), the mandatory base feature of purchasing a drink, present in all variants; `Tea` (t, in red), for serving tea; `Soda` (s, in green), for serving soda, which is a mandatory feature present in all variants; `CancelPurchase` (c, in brown), for canceling a purchase after a coin is entered; and `FreeDrinks` (f, in blue) for offering free drinks. Each transition is labeled by an *action* followed by a *feature expression*. For instance, the transition ① $\xrightarrow{\mathit{free}/f}$ ③ is included in variants where the feature f is enabled.

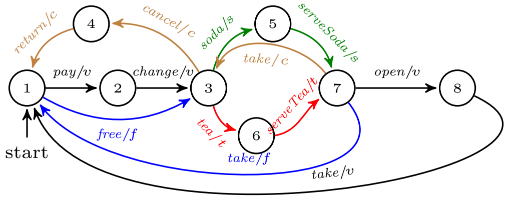

Fig. 1: The FTS for VendingMachine.

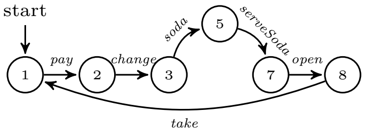

Fig. 2: $\pi_{\{v,s\}}$(VendingMachine)

By combining various features, a number of variants of this VendingMachine can be obtained. Recall that v and s are mandatory features. The set of valid configurations is thus: $\mathbb{K}^{\mathrm{VM}} = \{\{v,s\},\{v,s,t\},\{v,s,c\},\{v,s,t,c\},\{v,s,f\},\{v,s,t,f\},\{v,s,c,f\},\{v,s,t,c,f\}\}$. In Fig. 2 is shown the basic version of VendingMachine that only serves soda, which is described by the configuration: $\{v,s\}$ (or, as formula $v \wedge s \wedge \neg t \wedge \neg c \wedge \neg f$), that is the projection $\pi_{\{v,s\}}$(VendingMachine). It takes a coin, returns change, serves soda, opens a compartment so that the customer can take the soda, before closing it again.

Figures 3 and 8 show an MTS and an MFTS, respectively. Must transitions are denoted by solid lines, may transitions by dashed lines. The MFTS in Fig. 8 (Appendix B) has $\mathbb{F} = \{c\}$ and $\mathbb{K} = \{c, \neg c\}$. □

CTL* Properties. Computation Tree Logic* (CTL*) [2] is an expressive temporal logic for specifying system properties, which subsumes both CTL and LTL logics. CTL* state formulae Φ are generated by the following grammar:

$$\Phi ::= true \mid a \in AP \mid \neg a \mid \Phi_1 \wedge \Phi_2 \mid \forall \phi \mid \exists \phi, \qquad \phi ::= \Phi \mid \phi_1 \wedge \phi_2 \mid \bigcirc \phi \mid \phi_1 \mathsf{U} \phi_2$$

where ϕ represent CTL* path formulae. Note that the CTL* state formulae Φ are given in negation normal form ($\neg$ is applied only to atomic propositions). Given $\Phi \in$ CTL*, we consider $\neg\Phi$ to be the equivalent CTL* formula given in negation normal form. Other derived temporal operators (path formulae) can be defined as well by means of syntactic sugar, for instance: $\Diamond \phi = true \, \mathsf{U} \phi$ (ϕ holds eventually), and $\Box \phi = \neg \forall \Diamond \neg \phi$ (ϕ always holds). $\forall$CTL* and $\exists$CTL* are subsets of CTL* where the only allowed path quantifiers are $\forall$ and $\exists$, respectively.

We formalise the semantics of CTL* over a TS $\mathcal{T}$. We write $[\![\mathcal{T}]\!]^s_{\mathrm{TS}}$ for the set of executions that start in state s; $\rho[i] = s_i$ to denote the i-th state of the execution ρ; and $\rho_i = s_i \lambda_{i+1} s_{i+1} \ldots$ for the suffix of ρ starting from its i-th state.

Definition 5. *Satisfaction of a state formula Φ in a state s of a TS $\mathcal{T}$, denoted $\mathcal{T}, s \models \phi$, is defined as ($\mathcal{T}$ is omitted when clear from context):*

(1) $s \models a$ *iff* $a \in L(s)$; $s \models \neg a$ *iff* $a \notin L(s)$,
(2) $s \models \Phi_1 \wedge \Phi_2$ *iff* $s \models \Phi_1$ *and* $s \models \Phi_2$,
(3) $s \models \forall \phi$ *iff* $\forall \rho \in [\![\mathcal{T}]\!]^s_{TS}.\, \rho \models \phi$; $s \models \exists \phi$ *iff* $\exists \rho \in [\![\mathcal{T}]\!]^s_{TS}.\, \rho \models \phi$

Satisfaction of a path formula ϕ for an execution ρ of a TS $\mathcal{T}$, denoted $\mathcal{T}, \rho \models \phi$, is defined as ($\mathcal{T}$ is omitted when clear from context):

(4) $\rho \models \Phi$ *iff* $\rho[0] \models \Phi$,

(5) $\rho \models \phi_1 \wedge \phi_2$ *iff* $\rho \models \phi_1$ *and* $\rho \models \phi_2$; $\rho \models \bigcirc\phi$ *iff* $\rho_1 \models \phi$; $\rho \models (\phi_1 U \phi_2)$ *iff* $\exists i \geq 0. \big(\rho_i \models \phi_2 \wedge (\forall 0 \leq j \leq i-1. \, \rho_j \models \phi_1)\big)$

A TS $\mathcal{T}$ satisfies a state formula Φ, written $\mathcal{T} \models \Phi$, iff $\forall s_0 \in I. \, s_0 \models \Phi$.

Definition 6. *An FTS $\mathcal{F}$ satisfies a CTL* formula Φ, written $\mathcal{F} \models \Phi$, iff all its valid variants satisfy the formula: $\forall k \in \mathbb{K}. \, \pi_k(\mathcal{F}) \models \Phi$.*

The interpretation of CTL* over an MTS $\mathcal{M}$ is defined slightly different from the above Definition 5. In particular, the clause (3) is replaced by:

(3') $s \models \forall\phi$ iff for every may-execution ρ in the state s of $\mathcal{M}$, that is $\forall \rho \in [\![\mathcal{M}]\!]_{MTS}^{may,s}$, it holds $\rho \models \phi$; whereas $s \models \exists\phi$ iff there exists a must-execution ρ in the state s of $\mathcal{M}$, that is $\exists \rho \in [\![\mathcal{M}]\!]_{MTS}^{must,s}$, such that $\rho \models \phi$.

From now on, we implicitly assume this adapted definition when interpreting CTL* formulae over MTSs and MFTSs.

Example 2. Consider the FTS VendingMachine in Fig. 1. Suppose that the proposition `start` holds in the initial state ①. An example property Φ_1 is: $\forall\Box\,\forall\Diamond$`start`, which states that in every state along every execution all possible continuations will eventually reach the initial state. This formula is in $\forall$CTL*. Note that VendingMachine $\not\models \Phi_1$. For example, if the feature c (`Cancel`) is enabled, a counter-example where the state ① is never reached is: ① → ③ → ⑤ → ⑦ → ③ → .... The set of violating products is $[\![c]\!] = \{\{v, s, c\}, \{v, s, t, c\}, \{v, s, c, f\}, \{v, s, t, c, f\}\} \subseteq \mathbb{K}^{\text{VM}}$. However, $\pi_{[\![\neg c]\!]}(\text{VendingMachine}) \models \Phi_1$.

Consider the property Φ_2: $\forall\Box\,\exists\Diamond$`start`, which describes a situation where in every state along every execution there exists a possible continuation that will eventually reach the `start` state. This is a CTL* formula, which is neither in $\forall$CTL* nor in $\exists$CTL*. Note that VendingMachine $\models \Phi_2$, since even for variants with the feature c there is a continuation from the state ③ back to ①.

Consider the $\exists$CTL* property Φ_3: $\exists\Box\,\exists\Diamond$`start`, which states that there exists an execution such that in every state along it there exists a possible continuation that will eventually reach the `start` state. The witnesses are ① → ② → ③ → ⑤ → ⑦ → ⑧ → ① ... for variants that satisfy $\neg c$, and ① → ③ → ⑤ → ⑦ → ③ → ④ → ① ... for variants with c. □

3 Abstraction of FTSs

We now introduce the variability abstractions which preserve full CTL and its universal and existential properties. They simplify the configuration space of an FTSs, by reducing the number of configurations and manipulating presence conditions of transitions. We start working with Galois connections [1] between Boolean complete lattices of feature expressions, and then induce a notion of abstraction of FTSs. We define two classes of abstractions. We use the standard conservative abstractions [16,17] as an instrument to eliminate variability from the FTS in an *over-approximating* way, so by adding more executions. We use the dual abstractions, which can also eliminate variability but through *under-approximating* the given FTS, so by dropping executions.

Domains. The Boolean complete lattice of feature expressions (propositional formulae over $\mathbb{F}$) is: $(\mathit{FeatExp}(\mathbb{F})_{/\equiv}, \models, \vee, \wedge, \mathit{true}, \mathit{false}, \neg)$. The elements of the domain $\mathit{FeatExp}(\mathbb{F})_{/\equiv}$ are equivalence classes of propositional formulae $\psi \in \mathit{FeatExp}(\mathbb{F})$ obtained by quotienting by the semantic equivalence $\equiv$. The ordering $\models$ is the standard entailment between propositional logics formulae, whereas the least upper bound and the greatest lower bound are just logical disjunction and conjunction respectively. Finally, the constant *false* is the least, *true* is the greatest element, and negation is the complement operator.

Conservative abstractions. The *join abstraction*, $\boldsymbol{\alpha}^{\mathrm{join}}$, merges 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 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: $\boldsymbol{\alpha}^{\mathrm{join}}(\mathbb{F}) = \emptyset$, and the abstract set of valid configurations is a singleton: $\boldsymbol{\alpha}^{\mathrm{join}}(\mathbb{K}) = \{\mathit{true}\}$ if $\mathbb{K} \neq \emptyset$. The abstraction and concretization functions between $\mathit{FeatExp}(\mathbb{F})$ and $\mathit{FeatExp}(\emptyset)$, forming a Galois connection [16,17], are defined as:

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

The *feature ignore abstraction*, $\boldsymbol{\alpha}_A^{\mathrm{fignore}}$, introduces an over-approximation by ignoring a single feature $A \in \mathbb{F}$. It merges 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. The features and configurations of the abstracted model are: $\boldsymbol{\alpha}_A^{\mathrm{fignore}}(\mathbb{F}) = \mathbb{F} \setminus \{A\}$, and $\boldsymbol{\alpha}_A^{\mathrm{fignore}}(\mathbb{K}) = \{k[l_A \mapsto \mathit{true}] \mid k \in \mathbb{K}\}$, where l_A denotes a literal of A (either A or $\neg A$), and $k[l_A \mapsto \mathit{true}]$ is a formula resulting from k by substituting *true* for l_A. The abstraction and concretization functions between $\mathit{FeatExp}(\mathbb{F})$ and $\mathit{FeatExp}(\boldsymbol{\alpha}_A^{\mathrm{fignore}}(\mathbb{F}))$, forming a Galois connection [16,17], are:

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

where ψ and ψ' need to be in negation normal form before substitution.

[1] $\langle L, \leq_L \rangle \overset{\gamma}{\underset{\alpha}{\leftrightarrows}} \langle M, \leq_M \rangle$ is a *Galois connection* between complete lattices L (concrete domain) and M (abstract domain) iff $\alpha : L \to M$ and $\gamma : M \to L$ 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. We will often simply write (α, γ) for any such Galois connection.

Dual abstractions. Suppose that $\langle \mathit{FeatExp}(\mathbb{F})_{/\equiv}, \models\rangle$, $\langle \mathit{FeatExp}(\alpha(\mathbb{F}))_{/\equiv}, \models\rangle$ are Boolean complete lattices, and $\langle \mathit{FeatExp}(\mathbb{F})_{/\equiv}, \models\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathit{FeatExp}(\alpha(\mathbb{F}))_{/\equiv}, \models\rangle$ is a Galois connection. We define [11]: $\widetilde{\alpha} = \neg \circ \alpha \circ \neg$ and $\widetilde{\gamma} = \neg \circ \gamma \circ \neg$ so that $\langle \mathit{FeatExp}(\mathbb{F})_{/\equiv}, \Dashv\rangle \underset{\widetilde{\alpha}}{\overset{\widetilde{\gamma}}{\leftrightarrows}} \langle \mathit{FeatExp}(\alpha(\mathbb{F}))_{/\equiv}, \Dashv\rangle$ is a Galois connection (or equivalently, $\langle \mathit{FeatExp}(\alpha(\mathbb{F}))_{/\equiv}, \models\rangle \underset{\widetilde{\gamma}}{\overset{\widetilde{\alpha}}{\leftrightarrows}} \langle \mathit{FeatExp}(\mathbb{F})_{/\equiv}, \models\rangle$). The obtained Galois connections $(\widetilde{\alpha}, \widetilde{\gamma})$ are called dual (under-approximating) abstractions of (α, γ).

The *dual join abstraction*, $\widetilde{\boldsymbol{\alpha}^{\text{join}}}$, merges the control-flow of all variants, obtaining a single variant that includes only those executions that occur in all variants. Each feature expression ψ is replaced with *true* if all configurations from $\mathbb{K}$ satisfy ψ. The abstraction and concretization functions between $\mathit{FeatExp}(\mathbb{F})$ and $\mathit{FeatExp}(\emptyset)$, forming a Galois connection, are defined as: $\widetilde{\boldsymbol{\alpha}^{\text{join}}} = \neg \circ \boldsymbol{\alpha}^{\text{join}} \circ \neg$ and $\widetilde{\boldsymbol{\gamma}^{\text{join}}} = \neg \circ \boldsymbol{\gamma}^{\text{join}} \circ \neg$, that is:

$$\widetilde{\boldsymbol{\alpha}^{\text{join}}}(\psi) = \begin{cases} \mathit{true} & \text{if } \forall k \in \mathbb{K}. k \models \psi \\ \mathit{false} & \text{otherwise} \end{cases} \qquad \widetilde{\boldsymbol{\gamma}^{\text{join}}}(\psi) = \begin{cases} \bigwedge_{k \in 2^{\mathbb{F}} \setminus \mathbb{K}} (\neg k) & \text{if } \psi \text{ is } \mathit{true} \\ \mathit{false} & \text{if } \psi \text{ is } \mathit{false} \end{cases}$$

The *dual feature ignore abstraction*, $\widetilde{\boldsymbol{\alpha}_A^{\text{fignore}}}$, introduces an under-approximation by ignoring the feature $A \in \mathbb{F}$, such that the literals of A (that is, A and $\neg A$) are replaced with *false* in feature expressions (given in negation normal form). The abstraction and concretization functions between $\mathit{FeatExp}(\mathbb{F})$ and $\mathit{FeatExp}(\boldsymbol{\alpha}_A^{\text{fignore}}(\mathbb{F}))$, forming a Galois connection, are defined as: $\widetilde{\boldsymbol{\alpha}_A^{\text{fignore}}} = \neg \circ \boldsymbol{\alpha}_A^{\text{fignore}} \circ \neg$ and $\widetilde{\boldsymbol{\gamma}_A^{\text{fignore}}} = \neg \circ \boldsymbol{\gamma}_A^{\text{fignore}} \circ \neg$, that is:

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

where ψ and ψ' are in negation normal form.

Abstract MFTS and Preservation of CTL*. Given a Galois connection (α, γ) defined on the level of feature expressions, we now define the abstraction of an FTS as an MFTS with two transition relations: one (may) preserving universal properties, and the other (must) existential properties. The may transitions describe the behaviour that is possible, but not need be realized in the variants of the family; whereas the must transitions describe behaviour that has to be present in any variant of the family.

Definition 7. *Given the FTS $\mathcal{F} = (S, \mathit{Act}, \mathit{trans}, I, \mathit{AP}, L, \mathbb{F}, \mathbb{K}, \delta)$, we define the MFTS $\alpha(\mathcal{F}) = (S, \mathit{Act}, \mathit{trans}^{may}, \mathit{trans}^{must}, I, \mathit{AP}, L, \alpha(\mathbb{F}), \alpha(\mathbb{K}), \delta^{may}, \delta^{must})$ to be its abstraction, where $\delta^{may}(t) = \alpha(\delta(t))$, $\delta^{must}(t) = \widetilde{\alpha}(\delta(t))$, $\mathit{trans}^{may} = \{t \in \mathit{trans} \mid \delta^{may}(t) \neq \mathit{false}\}$, and $\mathit{trans}^{must} = \{t \in \mathit{trans} \mid \delta^{must}(t) \neq \mathit{false}\}$.*

Note that the degree of reduction is determined by the choice of abstraction and may hence be arbitrary large. In the extreme case of join abstraction, we obtain an abstract model with no variability in it, that is $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F})$ is an ordinary MTS.

Example 3. Recall the FTS VendingMachine of Fig. 1 with the set of valid configurations $\mathbb{K}^{\text{VM}}$ (see Example 1). Fig. 3 shows $\boldsymbol{\alpha}^{\text{join}}(\textsc{VendingMachine})$, where the allowed (may) part of the behavior includes the transitions that are associated with the optional features c, f, t in VendingMachine, whereas the required (must) part includes the transitions associated with the mandatory features v and s. Note that $\boldsymbol{\alpha}^{\text{join}}(\textsc{VendingMachine})$ is an ordinary MTS with no variability. The MFTS $\boldsymbol{\alpha}^{\text{fignore}}_{\{t,f\}}(\pi_{[\![v \wedge s]\!]}(\textsc{VendingMachine}))$ is shown in Fig. 8 (Appendix B). It has the singleton set of features $\mathbb{F} = \{c\}$ and limited variability $\mathbb{K} = \{c, \neg c\}$, where the mandatory features v and s are enabled. □

From the MFTS (resp., MTS) $\mathcal{MF}$, we define two FTSs (resp., TSs) $\mathcal{MF}^{may}$ and $\mathcal{MF}^{must}$ representing the may- and must-components of $\mathcal{MF}$, i.e. its may and must transitions, respectively. Thus, we have $[\![\mathcal{MF}^{may}]\!]_{FTS} = [\![\mathcal{MF}]\!]^{may}_{MFTS}$ and $[\![\mathcal{MF}^{must}]\!]_{FTS} = [\![\mathcal{MF}]\!]^{must}_{MFTS}$.

We now show that the abstraction of an FTS is sound with respect to CTL*. First, we show two helper lemmas stating that: for any variant $k \in \mathbb{K}$ that can execute a behavior, there exists an abstract variant $k' \in \alpha(\mathbb{K})$ that executes the same may-behaviour; and for any abstract variant $k' \in \alpha(\mathbb{K})$ that can execute a must-behavior, there exists a variant $k \in \mathbb{K}$ that executes the same behaviour [2].

Lemma 1. *Let $\psi \in FeatExp(\mathbb{F})$, and $\mathbb{K}$ be a set of valid configurations over $\mathbb{F}$.*

(i) *Let $k \in \mathbb{K}$ and $k \models \psi$. Then there exists $k' \in \alpha(\mathbb{K})$, such that $k' \models \alpha(\psi)$.*
(ii) *Let $k' \in \alpha(\mathbb{K})$ and $k' \models \widetilde{\alpha}(\psi)$. Then there exists $k \in \mathbb{K}$, such that $k \models \psi$.*

Lemma 2. (i) *Let $k \in \mathbb{K}$ and $\rho \in [\![\pi_k(\mathcal{F})]\!]_{TS} \subseteq [\![\mathcal{F}]\!]_{FTS}$. Then there exists $k' \in \alpha(\mathbb{K})$, such that $\rho \in [\![\pi_{k'}(\alpha(\mathcal{F}))]\!]^{may}_{MTS} \subseteq [\![\alpha(\mathcal{F})]\!]^{may}_{MFTS}$ is a may-execution in $\alpha(\mathcal{F})$.*
(ii) *Let $k' \in \alpha(\mathbb{K})$ and $\rho \in [\![\pi_{k'}(\alpha(\mathcal{F}))]\!]^{must}_{MTS} \subseteq [\![\alpha(\mathcal{F})]\!]^{must}_{MFTS}$ be a must-execution in $\alpha(\mathcal{F})$. Then there exists $k \in \mathbb{K}$, such that $\rho \in [\![\pi_k(\mathcal{F})]\!]_{TS} \subseteq [\![\mathcal{F}]\!]_{FTS}$.*

As a result, every $\forall$CTL* (resp., $\exists$CTL*) property true for the may- (resp., must-) component of $\alpha(\mathcal{F})$ is true for $\mathcal{F}$ as well. Moreover, the MFTS $\alpha(\mathcal{F})$ preserves the full CTL*.

Theorem 1 (Preservation results). *For any FTS $\mathcal{F}$ and (α, γ), we have:*

($\forall$CTL*) *For every $\Phi \in \forall CTL^\star$, $\alpha(\mathcal{F})^{may} \models \Phi \implies \mathcal{F} \models \Phi$.*
($\exists$CTL*) *For every $\Phi \in \exists CTL^\star$, $\alpha(\mathcal{F})^{must} \models \Phi \implies \mathcal{F} \models \Phi$.*
(CTL*) *For every $\Phi \in CTL^\star$, $\alpha(\mathcal{F}) \models \Phi \implies \mathcal{F} \models \Phi$.*

Abstract models are designed to be conservative for the satisfaction of properties. However, in case of the refutation of a property, a counter-example is found in the abstract model which may be spurious (introduced due to abstraction) for some variants and genuine for the others. This can be established by checking which variants can execute the found counter-example.

Let Φ be a CTL* formula which is not in $\forall$CTL* nor in $\exists$CTL*, and let $\mathcal{MF}$ be an MFTS. We verify $\mathcal{MF} \models \Phi$ by checking Φ on two FTSs $\mathcal{MF}^{may}$ and $\mathcal{MF}^{must}$, and then we combine the obtained results as specified below.

[2] Proofs of all lemmas and theorems in this section can be found in Appendix A.

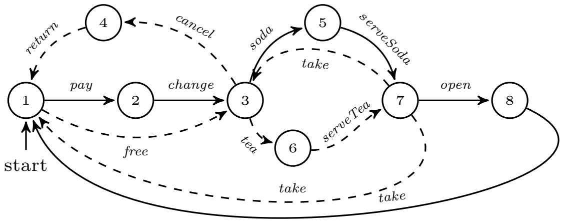

Fig. 3: $\boldsymbol{\alpha}^{\text{join}}(\textsc{VendingMachine})$.

Theorem 2. *For every $\Phi \in CTL^\star$ and MFTS $\mathcal{MF}$, we have:*

$$\mathcal{MF} \models \Phi = \begin{cases} \mathit{true} & \mathit{if}\ \big(\mathcal{MF}^{may} \models \Phi \wedge \mathcal{MF}^{must} \models \Phi\big) \\ \mathit{false} & \mathit{if}\ \big(\mathcal{MF}^{may} \not\models \Phi \vee \mathcal{MF}^{must} \not\models \Phi\big) \end{cases}$$

Therefore, we can check a formula Φ which is not in $\forall$CTL* nor in $\exists$CTL* on $\alpha(\mathcal{F})$ by running a model checker twice, once with the may-component of $\alpha(\mathcal{F})$ and once with the must-component of $\alpha(\mathcal{F})$. On the other hand, a formula Φ from $\forall$CTL* (resp., $\exists$CTL*) on $\alpha(\mathcal{F})$ is checked by running a model checker only once with the may-component (resp., must-component) of $\alpha(\mathcal{F})$.

The family-based model checking problem can be reduced to a number of smaller problems by partitioning the set of variants. Let the subsets $\mathbb{K}_1, \mathbb{K}_2, \ldots, \mathbb{K}_n$ form a *partition* of the set $\mathbb{K}$. Then: $\mathcal{F} \models \Phi$ iff $\pi_{\mathbb{K}_i}(\mathcal{F}) \models \Phi$ for all $i = 1, \ldots, n$. By using Theorem 1 (CTL*), we obtain the following result.

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

Therefore, in case of suitable partitioning of $\mathbb{K}$ and the aggressive $\boldsymbol{\alpha}^{\text{join}}$ abstraction, all $\boldsymbol{\alpha}^{\text{join}}(\pi_{\mathbb{K}_i}(\mathcal{F}))^{may}$ and $\boldsymbol{\alpha}^{\text{join}}(\pi_{\mathbb{K}_i}(\mathcal{F}))^{must}$ are ordinary TSs, so the family-based model checking problem can be solved using existing single-system model checkers with all the optimizations that these tools may already implement.

Example 4. Consider the properties introduced in Example 2. Using the TS $\boldsymbol{\alpha}^{\text{join}}(\textsc{VendingMachine})^{may}$ we can verify $\Phi_1 = \forall\Box\,\forall\Diamond\texttt{start}$ (Theorem 1, ($\forall$CTL*)). We obtain the counter-example ① → ③ → ⑤ → ⑦ → ③ . . ., which is genuine for variants satisfying c. Hence, variants from $[\![c]\!]$ violate Φ_1. On the other hand, by verifying that $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg c]\!]}(\textsc{VendingMachine}))^{may}$ satisfies Φ_1, we can conclude by Theorem 1, ($\forall$CTL*) that variants from $[\![\neg c]\!]$ satisfy Φ_1.

We can verify $\Phi_2 = \forall\Box\,\exists\Diamond\texttt{start}$ by checking may- and must-components of $\boldsymbol{\alpha}^{\text{join}}(\textsc{VendingMachine})$. In particular, we have $\boldsymbol{\alpha}^{\text{join}}(\textsc{VendingMachine})^{may} \models \Phi_2$ and $\boldsymbol{\alpha}^{\text{join}}(\textsc{VendingMachine})^{must} \models \Phi_2$. Thus, using Theorem 1, (CTL*) and Theorem 2, we have that $\textsc{VendingMachine} \models \Phi_2$.

Using $\boldsymbol{\alpha}^{\text{join}}(\textsc{VendingMachine})^{must}$ we can verify $\Phi_3 = \exists\Box\,\exists\Diamond\texttt{start}$, by finding the witness ① → ② → ③ → ⑤ → ⑦ → ⑧ → ① . . .. By Theorem 1, ($\exists$CTL*), we have that $\textsc{VendingMachine} \models \Phi_3$. □

4 Implementation

We now describe an implementation of our abstraction-based approach for CTL model checking of variational systems in the context of the state-of-the-art NuSMV model checker [5]. Since it is difficult to use FTSs to directly model very large variational systems, we use a high-level modelling language, called fNuSMV, which is expressively equivalent to FTSs and close to NuSMV's input language. Then, we show how to implement projection and variability abstractions as syntactic transformations of fNuSMV models.

A High-level Modelling Language. fNuSMV is a feature-oriented extension of the input language of NuSMV, which was introduced by Plath and Ryan [31] and subsequently improved by Classen [6]. A NuSMV model consists of a set of variable declarations and a set of assignments. The variable declarations define the state space and the assignments define the transition relation of the finite state machine described by the given model. For each variable, there are assignments that define its initial value and its value in the next state, which is given as a function of the variable values in the present state. Modules can be used to encapsulate and factor out recurring submodels. Consider a basic NuSMV model shown in Fig. 4a. It consists of a single variable x which is initialized to 0 and does not change its value. The property (marked by the keyword `SPEC`) is "$\forall\Diamond(x \geq k)$", where k is a meta-variable that can be replaced with various natural numbers. For this model, the property holds when $k = 0$. In all other cases (for $k > 0$), a counterexample is reported where x stays 0.

The fNuSMV language [31] is based on superimposition. *Features* are modelled as self-contained textual units using a new `FEATURE` construct added to the NuSMV language. A feature describes the changes to be made to the given basic NuSMV model. It can introduce new variables into the system (in a section marked by the keyword `INTRODUCE`), override the definition of existing variables in the basic model and change the values of those variables when they are read (in a section marked by the keyword `CHANGE`). For example, Fig. 4b shows a `FEATURE` construct, called A, which changes the basic model in Fig. 4a. In particular, the feature A defines a new variable nA initialized to 0. The basic system is changed in such a way that when the condition "$nA = 0$" holds then in the next state the basic system's variable x is incremented by 1 and in this case (when x is incremented) nA is set to 1. Otherwise, the basic system is not changed.

Classen [6] shows that fNuSMV and FTS are expressively equivalent. He [6] also proposes a way of composing fNuSMV features with the basic model to create a single model in pure NuSMV which describes all valid variants. The information about the variability and features in the composed model is recorded in the states. This is a slight deviation from the encoding in FTSs, where this information is part of the transition relation. However, this encoding has the advantage of being implementable in NuSMV without drastic changes to the model checker and its input language. Basically, in the composed model each feature becomes a Boolean state variable, which is non-deterministically initialised and whose value never changes. Thus, the initial states of the composed

```
MODULE main
VAR x : 0..1;
ASSIGN
  init(x) := 0;
  next(x) := x;
SPEC AF(x ≥ k);
```

(a) The basic model.

```
FEATURE A
INTRODUCE
  VAR nA : 0..1;
  ASSIGN init(nA) := 0;
CHANGE
  IF (nA = 0) THEN
  IMPOSE next(x) := x + 1;
           next(nA) :=
        next(x)=x+1?1:nA;
```

(b) The feature A.

```
MODULE features
  VAR fA : boolean;
  ASSIGN
    init(fA) := {TRUE,FALSE};
    next(fA) := fA;
MODULE main
  VAR f : features; x : 0..1; nA : 0..1;
  ASSIGN
     init(x) := 0; init(nA) := 0;
     next(x) := case f.fA & nA=0 : x+1;
                     TRUE : x;
                easc;
     next(nA) := case
          f.fA & nA=0 & next(x)=x+1 : 1;
          TRUE : nA;
                  easc;
```

(c) The composed model $\mathcal{M}$.

Fig. 4: NuSMV models.

model include all possible feature combinations. Every change performed by a feature in the composition is guarded by the corresponding feature variable.

For example, the composition of the basic model and the feature A given in Figs. 4a and 4b results in the model shown in Fig. 4c. First, a module, called *features*, containing all features (in this case, the single one A) is added to the system. To each feature (e.g. A) corresponds one variable in this module (e.g. fA). The *main* module contains a variable named f of type *features*, so that all feature variables can be referenced in it (e.g. $f.fA$). In the next state, the variable x is incremented by 1 when the feature A is enabled (fA is *TRUE*) and nA is 0. Otherwise (*TRUE:* can be read as *else:*), x is not changed. Also, nA is set to 1 when A is enabled and x is incremented by 1. The property $\forall\Diamond(x \geq 0)$ holds for both variants when A is enabled and A is disabled (fA is *FALSE*).

Transformations. We present the syntactic transformations of fNuSMV models defined by projection and variability abstractions. Let M represent a model obtained by composing a basic model with a set of features $\mathbb{F}$. Let M contain a set of assignments of the form: $s(v) := \texttt{case}\ b_1 : e_1; \dots b_n : e_n;\ \texttt{esac}$, where v is a variable, b_i is a boolean expression, e_i is an expression (for $1 \leq i \leq n$), and $s(v)$ is one of v, $\texttt{init}(v)$, or $\texttt{next}(v)$. We denote by $[\![M]\!]$ the FTS for this model [6].

Let $\mathbb{K}' \subseteq 2^{\mathbb{F}}$ be a set of configurations described by a feature expression ψ', i.e. $[\![\psi']\!] = \mathbb{K}'$. The projection $\pi_{[\![\psi']\!]}([\![M]\!])$ is obtained by adding the INVAR constraint ψ' to the model M, denoted as $M + \texttt{INVAR}(\psi')$. Thus, $\pi_{[\![\psi']\!]}([\![M]\!]) = [\![M + \texttt{INVAR}(\psi')]\!]$. Another solution would be to add the constraint ψ' to each b_i in the assignments to the state variables.

```
MODULE main
  VAR x : 0..1; nA : 0..1; rnd : boolean;
  ASSIGN
    init(x) := 0; init(nA) := 0;
    init(rnd) := {TRUE,FALSE};
    next(x) := case rnd & nA=0 : x + 1;
                    TRUE : x; easc;
    next(nA) := case
      rnd & nA=0 & next(x)=x+1 : 1;
      TRUE : nA; easc;
```

Fig. 5: $\boldsymbol{\alpha}^{\text{join}}(\mathcal{M})^{may}$

```
MODULE main
  VAR x : 0..1; nA : 0..1;
  ASSIGN
    init(x) := 0; init(nA) := 0;
    next(x) := x;
    next(nA) := nA;
```

Fig. 6: $\boldsymbol{\alpha}^{\text{join}}(\mathcal{M})^{must}$

Let (α, γ) be a Galois connection from Section 3. The abstract $\alpha(M)^{may}$ and $\alpha(M)^{must}$ are obtained by the following rewrites for assignments in M:

$$\alpha\big(s(v) := \texttt{case}\, b_1 : e_1; \ldots b_n : e_n; \texttt{esac}\big)^{may} = s(v) := \texttt{case}\, \alpha^m(b_1) : e_1; \ldots \alpha^m(b_n) : e_n; \texttt{esac}$$
$$\alpha\big(s(v) := \texttt{case}\; b_1 : e_1; \ldots b_n : e_n; \texttt{esac}\big)^{must} = s(v) := \texttt{case}\; \widetilde{\alpha}(b_1) : e_1; \ldots \widetilde{\alpha}(b_n) : e_n; \texttt{esac}$$

The functions α^m and $\widetilde{\alpha}$ copy all basic boolean expressions other than feature expressions, and recursively calls itself for all sub-expressions of compound expressions. For $\boldsymbol{\alpha}^{\text{join}}(M)^{may}$, we have a single Boolean variable *rnd* which is non-deterministically initialized. Then, $\alpha^m(\psi) = \mathit{rnd}$ if $\alpha(\psi) = \mathit{true}$. We have: $\alpha(\llbracket M \rrbracket)^{may} = \llbracket \alpha(M)^{may} \rrbracket$ and $\alpha(\llbracket M \rrbracket)^{must} = \llbracket \alpha(M)^{must} \rrbracket$. For example, given the composed model $\mathcal{M}$ in Fig. 4c, the abstractions $\boldsymbol{\alpha}^{\text{join}}(\mathcal{M})^{may}$ and $\boldsymbol{\alpha}^{\text{join}}(\mathcal{M})^{must}$ are shown in Figs. 5 and 6, respectively. Note that $\widetilde{\boldsymbol{\alpha}^{\text{join}}}(f.fA) = \mathit{false}$, so the first branch of `case` statements in $\mathcal{M}$ is never taken in $\boldsymbol{\alpha}^{\text{join}}(\mathcal{M})^{must}$.

5 Evaluation

We now evaluate our abstraction-based verification technique. First, we show how variability abstractions can turn a previously infeasible analysis of variability model into a feasible one. Second, we show that instead of verifying CTL properties using the family-based version of NuSMV[3], we can use variability abstraction to obtain an abstract variability model (with a low number of variants) that can be subsequently model checked using the standard version of NuSMV.

All experiments were executed on a 64-bit Intel®Core^{TM} i7-4600U CPU running at 2.10 GHz with 8 GB memory. The implementation, benchmarks, and all results obtained from our experiments are available from: `https://aleksdimovski.github.io/abstract-ctl.html`. The reported performance numbers constitute the average runtime of five independent executions. For each experiment, we report the time needed to perform the verification task in seconds. The BDD model checker NuSMV is run with the parameter `-df -dynamic`, which ensures that

[3] An extended version of NuSMV [9] implements the family-based algorithm for variational models obtained by composing the basic model and all available features.

prop-	*family-based app.*		*abstraction-based app.*		*improvement*
-erty	$\lvert\mathbb{K}\rvert$	Time	$\lvert\alpha(\mathbb{K})\rvert$	Time	Time
$\varPhi_1$	512	36.73 s	2	2.59 s	14 ×
$\varPhi_2$	512	35.89 s	2	6.95 s	5 ×
$\varPhi_3$	512	54.76 s	1	1.67 s	32 ×
$\varPhi_4$	512	2.65 s	2	1.04 s	2.5 ×
$\varPhi_5$	512	37.76 s	2	2.62 s	15 ×

Fig. 7: Verification of Elevator properties using tailored abstractions. We compare family-based approach vs. abstraction-based approach.

the BDD package reorders the variables during verification in case the BDD size grows beyond a certain threshold. We consider two case studies: a synthetic example to demonstrate specific characteristics of our approach, and the Elevator system [31] which is a standard benchmark in the SPLE community [9,6,7,17].

Synthetic example. As an experiment, we have tested limits of family-based model checking with extended NuSMV and "brute-force" single-system model checking with standard NuSMV (where all variants are verified one by one). We have gradually added variability to the variational model in Fig. 4. This was done by adding optional features which increase the basic model's variable x by the number corresponding to the given feature. For example, the `CHANGE` section for the second feature B is: `IF` $(nB = 0)$ `THEN IMPOSE next`$(x) := x{+}2$; `next`$(nB) :=$ `next`$(x){=}x{+}2?1{:}nB$, and the domain of x is 0..3.

We check the assertion $\forall\Diamond(x \geq 0)$. For $|\mathbb{F}| = 25$ (for which $|\mathbb{K}| = 2^{25}$ variants, and the state space is 2^{32}) the family-based NuSMV takes around 77 minutes to verify the assertion, whereas for $|\mathbb{F}| = 26$ it has not finished the task within two hours. The analysis time to check the assertion using "brute force" with standard NuSMV ascends to almost three years for $|\mathbb{F}| = 25$. On the other hand, if we apply the variability abstraction $\boldsymbol{\alpha}^{\text{join}}$, we are able to verify the same assertion by only one call to standard NuSMV on the *abstracted* model in 2.54 seconds for $|\mathbb{F}| = 25$ and in 2.99 seconds for $|\mathbb{F}| = 26$.

Elevator*.* The Elevator, designed by Plath and Ryan [31], contains about 300 LOC and 9 independent features: `Antiprunk`, `Empty`, `Exec`, `OpenIfIdle`, `Overload`, `Park`, `QuickClose`, `Shuttle`, and `TTFull`, thus yielding $2^9 = 512$ variants. The elevator serves a number of floors (which is five in our case) such that there is a single platform button on each floor which calls the elevator. The elevator will always serve all requests in its current direction before it stops and changes direction. When serving a floor, the elevator door opens and closes again. The size of the Elevator model is 2^{28} states. On the other hand, the sizes of $\boldsymbol{\alpha}^{\text{join}}(\text{Elevator})^{may}$ and $\boldsymbol{\alpha}^{\text{join}}(\text{Elevator})^{must}$ are 2^{20} and 2^{19} states, resp.

We consider five properties. The $\forall$CTL property "$\varPhi_1 = \forall\Box(floor = 2 \land liftBut5.pressed \land direction = up \Rightarrow \forall[direction = up\, \mathsf{U}\, floor = 5]$" is that, when the elevator is on the second floor with direction up and the button five is pressed, then the elevator will go up until the fifth floor is reached. This property is violated by variants for which `Overload` (the elevator will refuse to close its doors when it is overloaded) is satisfied. Given sufficient knowledge of the system and the property, we can tailor an abstraction for verifying this property more effectively. We call standard NuSMV to check Φ_1 on two models $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\texttt{Overload}]\!]}(\textsc{Elevator}))^{may}$ and $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg\texttt{Overload}]\!]}(\textsc{Elevator}))^{may}$. For the first abstracted projection we obtain an "abstract" counter-example violating Φ_1, whereas the second abstracted projection satisfies Φ_1. Similarly, we can verify that the $\forall$CTL property "$\Phi_2 = \forall\Box\,(floor = 2 \wedge direction = up \Rightarrow \forall \bigcirc\,(direction = up))$" is satisfied only by variants with enabled `Shuttle` (the lift will change direction at the first and last floor). We can successfully verify Φ_2 for $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\texttt{Shuttle}]\!]}(\textsc{Elevator}))^{may}$ and obtain a counter-example for $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg\texttt{Shuttle}]\!]}(\textsc{Elevator}))^{may}$. The $\exists$CTL property "$\Phi_3 = (\texttt{OpenIfIdle} \wedge \neg\texttt{QuickClose}) \implies \exists\Diamond(\exists\Box\,(door = open))$" is that, there exists an execution such that from some state on the door stays open. We can invoke the standard NuSMV to verify that Φ_3 holds for $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\texttt{OpenIfIdle}\wedge\neg\texttt{QuickClose}]\!]}(\textsc{Elevator}))^{must}$. The following two properties are neither in $\forall$CTL nor in $\exists$CTL. The property "$\Phi_4 = \forall\Box\,(floor = 1 \wedge idle \wedge door = closed \implies \exists\Box(floor = 1 \wedge door = closed))$" is that, for any execution globally if the elevator is on the first floor, idle, and its door is closed, then there is a continuation where the elevator stays on the first floor with closed door. The satisfaction of Φ_4 can be established by verifying it against both $\boldsymbol{\alpha}^{\text{join}}(\textsc{Elevator})^{may}$ and $\boldsymbol{\alpha}^{\text{join}}(\textsc{Elevator})^{must}$ using two calls to standard NuSMV. The property "$\Phi_5 = \texttt{Park} \implies \forall\Box\,(floor = 1 \wedge idle \implies \exists[idle \mathsf{U} floor = 1])$" is satisfied by all variants with enabled `Park` (when idle, the elevator returns to the first floor). We can successfully verify Φ_5 by analyzing $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\texttt{Park}]\!]}(\textsc{Elevator}))^{may}$ and $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\texttt{Park}]\!]}(\textsc{Elevator}))^{must}$ using two calls to standard NuSMV. We can see in Fig. 7 that abstractions achieve significant speed-ups between 2.5 and 32 times faster than the family-based approach.

6 Related Work

Recently, many family-based techniques that work on the level of variational systems have been proposed. This includes family-based syntax checking [27,22], family-based type checking [26], family-based static program analysis [30,18,19], family-based verification [25,32,24], 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) [28,3]. Subsequently, Classen et al. present FTSs [8] and specifically designed family-based model checking algorithms for verifying FTSs against LTL [7]. This approach is extended [9,6] to enable verification of CTL properties using an family-based version of NuSMV. The work [4] shows how modal μ-calculus properties of variational systems can be verified using a general-purpose model checker mCRL2. In order to make this family-based approach more scalable, the works [23,17] propose applying conservative variability abstractions on FTSs for deriving abstract family-based model checking of LTL. An automatic abstraction refinement procedure for family-based model checking is then proposed in [21], which works until a genuine counterexample is found or the property satisfaction is shown for all variants in the family. The application of variability abstractions for verifying LTL and ∀CTL of real-time variational systems is described in [20]. The works [13,15] present an approach for family-based software model checking of `#ifdef`-based (second-order) program families using symbolic game semantics models [12,14].

7 Conclusion

We have proposed conservative (over-approximating) and their dual (under-approximating) variability abstractions to derive abstract family-based model checking that preserves the full CTL*. The evaluation confirms that interesting properties can be efficiently verified in this way. In this work, we assume that a suitable abstraction is manually generated before verification. If we want to make the whole verification procedure automatic, we need to develop an abstraction and refinement framework for CTL* properties similar to the one in [21] which is designed for LTL.

References

1. Apel, S., Batory, D.S., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines - Concepts and Implementation. Springer (2013), http://dx.doi.org/10.1007/978-3-642-37521-7
2. Baier, C., Katoen, J.: Principles of model checking. MIT Press (2008)
3. 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
4. ter Beek, M.H., de Vink, E.P., Willemse, T.A.C.: Family-based model checking with mcrl2. In: Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Proceedings. LNCS, vol. 10202, pp. 387–405 (2017), https://doi.org/10.1007/978-3-662-54494-5_23
5. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: Nusmv 2: An opensource tool for symbolic model checking. In: Computer Aided Verification, 14th International Conference, CAV 2002, Proceedings. LNCS, vol. 2404, pp. 359–364. Springer (2002), https://doi.org/10.1007/3-540-45657-0_29
6. Classen, A.: Ctl model checking for software product lines in nusmv. Technical Report, P-CS-TR SPLMC-00000002, University Of Namur pp. 1–17 (2011)
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. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A.: Symbolic model checking of software product lines. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011. pp. 321–330. ACM (2011), http://doi.acm.org/10.1145/1985793.1985838
10. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley (2001)
11. Cousot, P.: Partial completeness of abstract fixpoint checking. In: Abstraction, Reformulation, and Approximation, 4th International Symposium, SARA 2000, Proceedings. LNCS, vol. 1864, pp. 1–25. Springer (2000), https://doi.org/10.1007/3-540-44914-0_1
12. 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
13. 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)
14. Dimovski, A.S.: Probabilistic analysis based on symbolic game semantics and model counting. In: Proceedings Eighth International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2017. EPTCS, vol. 256, pp. 1–15 (2017)
15. Dimovski, A.S.: Verifying annotated program families using symbolic game semantics. Theor. Comput. Sci. 706, 35–53 (2018), https://doi.org/10.1016/j.tcs.2017.09.029
16. 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
17. Dimovski, A.S., Al-Sibahi, A.S., Brabrand, C., Wasowski, A.: Efficient family-based model checking via variability abstractions. STTT 19(5), 585–603 (2017), https://doi.org/10.1007/s10009-016-0425-2
18. 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
19. 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
20. Dimovski, A.S., Wasowski, A.: From transition systems to variability models and from lifted model checking back to UPPAAL. In: Models, Algorithms, Logics and Tools - Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday. LNCS, vol. 10460, pp. 249–268. Springer (2017), https://doi.org/10.1007/978-3-319-63121-9_13
21. Dimovski, A.S., Wasowski, A.: Variability-specific abstraction refinement for family-based model checking. In: Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Proceedings. LNCS, vol. 10202, pp. 406–423 (2017), http://dx.doi.org/10.1007/978-3-662-54494-5_24
22. 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

23. Holzmann, G.J.: The SPIN Model Checker - primer and reference manual. Addison-Wesley (2004)
24. 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
25. Iosif-Lazar, A.F., Melo, J., Dimovski, A.S., Brabrand, C., Wasowski, A.: Effective analysis of c programs by rewriting variability. Programming Journal 1(1), 1 (2017), https://doi.org/10.22152/programming-journal.org/2017/1/1
26. Kästner, C., Apel, S., Thüm, T., Saake, G.: Type checking annotation-based product lines. ACM Trans. Softw. Eng. Methodol. 21(3), 14 (2012)
27. 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
28. 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
29. Larsen, K.G., Thomsen, B.: A modal process logic. In: Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88). pp. 203–210. IEEE Computer Society (1988), http://dx.doi.org/10.1109/LICS.1988.5119
30. 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
31. Plath, M., Ryan, M.: Feature integration using a feature construct. Sci. Comput. Program. 41(1), 53–84 (2001), https://doi.org/10.1016/S0167-6423(00)00018-6
32. 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

A Proofs

Lemma 1. Let $\psi \in \mathit{FeatExp}(\mathbb{F})$, and $\mathbb{K}$ be a set of configurations over $\mathbb{F}$.

(i) Let $k \in \mathbb{K}$ and $k \models \psi$. Then there exists $k' \in \alpha(\mathbb{K})$, such that $k' \models \alpha(\psi)$.
(ii) Let $k' \in \alpha(\mathbb{K})$ and $k' \models \widetilde{\alpha}(\psi)$. Then there exists $k \in \mathbb{K}$, such that $k \models \psi$.

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

(i) The proof is similar to proof of Lemma 2 in [17].
(ii) **Case $\boldsymbol{\alpha}^{\text{join}}$:** By assumption, we have that $\mathbb{K} \neq \emptyset$, thus $\boldsymbol{\alpha}^{\text{join}}(\mathbb{K}) = \{\mathit{true}\}$. Since $\mathit{true} \models \widetilde{\boldsymbol{\alpha}^{\text{join}}}(\psi)$, it follows that $\widetilde{\boldsymbol{\alpha}^{\text{join}}}(\psi) = \mathit{true}$. This is the case only if for all $k \in \mathbb{K}$, it holds $k \models \psi$.
Case $\widetilde{\boldsymbol{\alpha}_A^{\text{fignore}}}$: By assumption, $k' = k[l_A \mapsto \mathit{true}] \in \boldsymbol{\alpha}_A^{\text{fignore}}(\mathbb{K})$ and $k' \models \widetilde{\boldsymbol{\alpha}_A^{\text{fignore}}}(\psi)$. That is, $k[l_A \mapsto \mathit{true}] \models \psi[l_A \mapsto \mathit{false}]$. Thus, we must have that $k \models \psi$.

□

Lemma 2.

(i) Let $k \in \mathbb{K}$ and $\rho \in [\![\pi_k(\mathcal{F})]\!]_{TS} \subseteq [\![\mathcal{F}]\!]_{FTS}$. Then there exists $k' \in \alpha(\mathbb{K})$, such that $\rho \in [\![\pi_{k'}(\alpha(\mathcal{F}))]\!]_{MTS}^{may} \subseteq [\![\alpha(\mathcal{F})]\!]_{MFTS}^{may}$ is a may-execution in it.
(ii) Let $k' \in \alpha(\mathbb{K})$ and $\rho \in [\![\pi_{k'}(\alpha(\mathcal{F}))]\!]_{MTS}^{must} \subseteq [\![\alpha(\mathcal{F})]\!]_{MFTS}^{must}$ be a must-execution in it. Then there exists $k \in \mathbb{K}$, such that $\rho \in [\![\pi_k(\mathcal{F})]\!]_{TS} \subseteq [\![\mathcal{F}]\!]_{FTS}$.

Proof (Lemma 2).

(i) Let $\rho = s_0\lambda_1 s_1\lambda_2 \ldots \in [\![\pi_k(\mathcal{F})]\!]_{TS}$. This means that for all transitions in ρ, $t_i = s_i \xrightarrow{\lambda_{i+1}} s_{i+1}$, we have that $k \models \delta(t_i)$ for all $i \geq 0$. By Lemma 1(i), we have that there exists $k' \in \alpha(\mathbb{K})$, such that $k' \models \alpha(\delta(t_i))$, i.e. $k' \models \delta^{may}(t_i)$, for all $i \geq 0$. Hence, we have $\rho \in [\![\pi_{k'}(\alpha(\mathcal{F}))]\!]_{MTS}^{may}$.
(ii) Let $\rho = s_0\lambda_1 s_1\lambda_2 \ldots \in [\![\pi_{k'}(\alpha(\mathcal{F}))]\!]_{MTS}^{must}$. This means that for all transitions in ρ, $t_i = s_i \xrightarrow{\lambda_{i+1}} s_{i+1}$, we have that $k' \models \widetilde{\alpha}(\delta(t_i))$, i.e. $k' \models \delta^{must}(t_i)$, for all $i \geq 0$. By Lemma 1(ii), we have that there exists $k \in \mathbb{K}$, such that $k \models \delta(t_i)$ for all $i \geq 0$. Hence, we have $\rho \in [\![\pi_k(\mathcal{F})]\!]_{TS}$.

□

Theorem 1.[Preservation of CTL*] $\alpha(\mathcal{F}) \models \Phi \implies \mathcal{F} \models \Phi$.

Proof (Theorem 1). We prove the most difficult case [CTL*]. By induction on the structure of Φ. We prove for state formulae Φ that if $\alpha(\mathcal{F}) \models \Phi$ (i.e. $\pi_{k'}(\alpha(\mathcal{F})) \models \Phi$ for some $k' \in \alpha(\mathbb{K})$), then $\mathcal{F} \models \Phi$ (i.e. there exists some $k \in \mathbb{K}$, such that $\pi_{k'}(\mathcal{F}) \models \Phi$). All cases except $\forall$ and $\exists$ quantifiers are straightforward.

For $\Phi = \forall\phi$, we proceed by contraposition. Assume $\mathcal{F} \not\models \forall\phi$. Then, there exist a configuration $k \in \mathbb{K}$ and an execution $\rho \in [\![\pi_k(\mathcal{F})]\!]_{TS}$ such that $\rho \not\models \phi$, i.e. $\rho \models \neg\phi$. By Lemma 2(i), we have that there exists $k' \in \alpha(\mathbb{K})$, such that $\rho \in [\![\pi_{k'}(\alpha(\mathcal{F}))]\!]_{MTS}^{may}$, and so $\alpha(\mathcal{F}) \not\models \forall\phi$.

For $\Phi = \exists\phi$. Assume $\alpha(\mathcal{F}) \models \exists\phi$. Then, there exist a configuration $k' \in \alpha(\mathbb{K})$, such that $\pi_{k'}(\alpha(\mathcal{F})) \models \exists\phi$. This means that there exists an execution $\rho \in [\![\pi_{k'}(\alpha(\mathcal{F}))]\!]^{must}_{MTS}$ such that $\rho \models \phi$. By Lemma 2(ii), we have that there exists $k \in \mathbb{K}$, such that $\rho \in [\![\pi_k(\mathcal{F})]\!]_{TS}$, and so $\mathcal{F} \models \exists\phi$. □

Theorem 2. For every $\Phi \in CTL^\star$ and MFTS $\mathcal{MF}$, we have:

$$\mathcal{MF} \models \Phi = \begin{cases} \mathit{true} & \text{if } \big(\mathcal{MF}^{may} \models \Phi \wedge \mathcal{MF}^{must} \models \Phi\big) \\ \mathit{false} & \text{if } \big(\mathcal{MF}^{may} \not\models \Phi \vee \mathcal{MF}^{must} \not\models \Phi\big) \end{cases}$$

Proof (Theorem 2). By induction on the structure of Φ. See Appendix A. All cases except $\forall$ and $\exists$ quantifiers are straightforward.

For $\Phi = \forall\phi$. Consider the first case, when $\mathcal{MF} \models \Phi = \mathit{true}$. Assume $\mathcal{MF}^{may} \models \forall\phi$. That is, for any may-execution ρ of $\mathcal{MF}$ we have $\rho \models \phi$. By Definition 5 (3'), we have $\mathcal{MF} \models \Phi$. Consider the second case, when $\mathcal{MF} \models \Phi = \mathit{false}$. Assume $\mathcal{MF}^{may} \not\models \forall\phi$. That is, there exists a may-execution ρ of $\mathcal{MF}$ such that $\rho \models \phi$. By Definition 5 (3'), we have $\mathcal{MF} \not\models \Phi$. Assume $\mathcal{MF}^{must} \not\models \forall\phi$. That is, there exists a must-execution ρ of $\mathcal{MF}$ such that $\rho \not\models \phi$. But ρ ia also a may-execution, so by Definition 5 (3'), we have $\mathcal{MF} \not\models \Phi$.

For $\Phi = \exists\phi$. Consider the first case, when $\mathcal{MF} \models \Phi = \mathit{true}$. Assume $\mathcal{MF}^{must} \models \exists\phi$. That is, there exists a must-execution ρ of $\mathcal{MF}$ such that $\rho \models \phi$. By Definition 5 (3'), we have $\mathcal{MF} \models \Phi$. Consider the second case, when $\mathcal{MF} \models \Phi = \mathit{false}$. Assume $\mathcal{MF}^{may} \not\models \exists\phi$. That is, for all may-executions ρ of $\mathcal{MF}$ we have $\rho \not\models \phi$. Since all must-executions are also may-executions, we have that all must-executions do not satisfy ϕ. By Definition 5 (3'), we have $\mathcal{MF} \not\models \Phi$. Assume $\mathcal{MF}^{must} \not\models \exists\phi$. That is, for all must-executions ρ of $\mathcal{MF}$ we have $\rho \not\models \phi$. By Definition 5 (3'), we have $\mathcal{MF} \not\models \Phi$.

B Figures

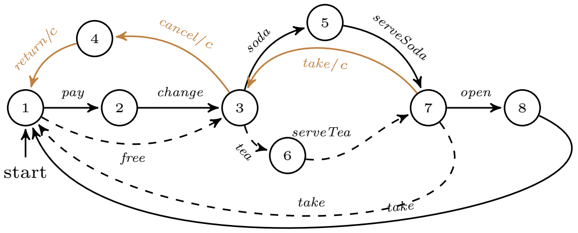

Fig. 8: $\boldsymbol{\alpha}^{\text{fignore}}_{\{t,f\}}(\pi_{[\![v \wedge s]\!]}(\textsc{VendingMachine}))$. For clarity, we omit to write the presence condition *true* in transitions.