$$\hbox {CTL}^{\star }$$ CTL # family-based model checking using variability abstractions and modal transition systems

Aleksandar S. Dimovski

International Journal on Software Tools for Technology Transfer

ISSN 1433-2779

Int J Softw Tools Technol Transfer
DOI 10.1007/s10009-019-00528-0

STTT

International Journal on
Software Tools for Technology Transfer

Springer

Springer

International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/s10009-019-00528-0

FASE 2018

CTL* family-based model checking using variability abstractions and modal transition systems

Aleksandar S. Dimovski[1]

Abstract

Variational systems can produce a (potentially huge) number of related systems, known as products or variants, by using features (configuration options) to mark the variable functionality. In many of the application domains, their rigorous verification and analysis are very important, yet the appropriate tools rarely are able to analyse variational systems. Recently, this problem was addressed by designing specialized so-called family-based model checking algorithms, which allow simultaneous verification of all variants in a single run by exploiting the commonalities between the variants. Yet, their computational cost still greatly depends on the number of variants (the size of configuration space), which is often huge. Moreover, their implementation and maintenance represent a costly research and development task. One of the most promising approaches to fighting the configuration space explosion problem is *variability abstractions*, which simplify variability away from variational systems. 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 transition system*, which preserves the satisfaction of both universal and existential temporal properties, as expressible in CTL*. Modal 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 configuration space to infer suitable divide-and-conquer verification plans for the given variational system. We illustrate the practicality of this approach for several variational systems using the standard version of (single-system) NuSMV model checker.

Keywords Software product line engineering · Family-based model checking · Abstract interpretation · Modal transition systems · Featured transition systems · CTL* temporal logic

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) [12], 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 domains (e.g. cars, phones, avionics, health care) [12,26]. In these domains, a rigorous verification and analysis are very important. Indeed, engineers have to provide solid proofs that all variants satisfy their desired properties. Among the methods included in current practices, *model checking* [3] is a well-studied technique used to establish or refute that temporal logic properties hold for a system.

Despite numerous benefits of variability and SPLE, a growing amount of variability leads to combinatorial complexity and, consequently, to severe challenges. Obviously, the size of the configuration space (i.e. the 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

---

✉ Aleksandar S. Dimovski
aleksandar.dimovski@unt.edu.mk

[1] Faculty of Informatics, Mother Teresa University, Skopje, Macedonia

variant is infeasible for realistic variational systems, due to the sheer number of variants. This is very ineffective also because the same execution behaviour is checked multiple times, whenever it is shared by several variants.

Researchers have addressed this problem by designing new more efficient verification techniques, which are based on using compact representations for modelling variational systems that incorporate the commonality within the family. We use the term *variability models* to denote such compact representations of variational systems. One of the most popular and widely accepted variability models today is *featured transition systems* (FTSs) [10]. Each behaviour in an FTS is associated with the set of variants able to produce it. A specialized family-based model checking algorithm [9,10] executed on such a model checks an execution behaviour only once regardless of how many variants include it. These algorithms model check all variants simultaneously in a single run and report precise conclusive results for all individual variants (whether they satisfy or violate a given property). Unfortunately, their performance *still* heavily depends on the size and complexity of the configuration space of the analysed variational system. Moreover, developing and 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 (FTS) 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 concrete configurations (variants) become indistinguishable and can be collapsed into a single abstract configuration (i.e. abstract variant). The abstract model is constructed in such a way that if some property holds for this abstract model, then it will also hold for the concrete variability model. The technique proposed in this paper significantly extends the scope of existing over-approximating variability abstractions introduced in [17,25], which support the verification of universal properties only (LTL and $\forall$CTL). More specifically, here we construct abstract variability models which can be used to check arbitrary formulae of CTL*, thus including arbitrary nested (universal and existential) path quantifiers. We use modal transition systems (MTSs) for representing abstract variability models. MTSs are transition systems (TSs) with two kinds of transitions, *must* and *may*, expressing behaviours that necessarily occur (must) or possibly occur (may) in the concrete model. We use the standard conservative (over-approximating) abstractions to define may-transitions, and their dual (under-approximating) abstractions to define must-transitions. Therefore, MTSs perform both over- and under-approximation, admitting both universal and existential properties to be deduced. Since the constructed abstract models (given as MTSs) preserve satisfiability of all CTL* properties, if any such property is true for an abstract MTS, then it is also true for the concrete variability model (which is given as an FTS). Moreover, we show that any model checking problem on MTSs can be reduced to two traditional model checking problems on standard TSs. In particular, we show that a property holds on an MTS, if both its may- and must-parts (which represent standard TSs) satisfy that property. The overall verification technique relies on partitioning the configuration space and abstracting separately concrete FTSs corresponding to individual partitions, until the point we obtain abstract models with no variability, so that it is feasible to complete their model checking in the brute-force fashion using the standard single-system model checkers.

The main goal of abstraction is to avoid the construction of the full concrete model. Hence, we want to derive an abstract model directly from some high-level description of the variational system. We show how partitionings and abstractions of variational systems specified in a high-level modelling language (which is an input language for the NuSMV model checker) can be implemented as simple preprocessor transformations. In this way, we do not perform any modifications to the model checker. Although the proposed technique is theoretically designed and shown correct for verifying arbitrarily CTL* properties, our implementation is based on the NuSMV model checker and CTL properties. This is due to the fact that we want to compare our technique with the traditional CTL family-based model checking algorithm (used as a baseline) [11], which is based on an extended version of NuSMV model checker and CTL properties. Experiments show that our technique combined with the standard version of NuSMV achieves considerable performance gains.

We make the following contributions here:

- Conservative and their dual *variability abstractions* for featured transition systems are defined.
- *Modal transition systems* for representing abstract variability models are introduced, and their soundness with respect to CTL* properties is shown.
- *Abstractions are implemented as syntactic transformations* of variational systems specified in a high-level modelling language in a preprocessing step, without any modifications to the model checking tools.
- *Efficient CTL family-based model checking* technique using an off-the-shelf model checker NuSMV is proposed.
- *Evaluation* of the above technique for CTL family-based model checking with NuSMV is presented, which shows scalability gains against the traditional CTL family-based model checking algorithms and against brute-force enumeration approach.

This work is an extended and revised version of [16]. We revise and correct the syntax and semantics of CTL* formulae in negation normal form given in [16]. We also make the following extensions here: (1) We motivate the need for using over- and under-approximating variability abstractions for achieving efficient CTL* family-based model checking; (2) we provide formal proofs for all main results in the work; (3) we expand and elaborate the examples as well as the discussion on how this approach works; (4) we provide more details on how may- and must-parts of high-level abstract models are defined as source-to-source transformations; (5) we greatly augment the evaluation of this approach by defining precise research questions, including new case-study models, considering more properties, and extending performance results.

We proceed with a motivating example for using variability abstractions for CTL* family-based model checking in Sect. 2. The basics of CTL* family-based model checking are explained in Sect. 3. Section 4 defines variability abstractions as well as abstract variability models and proves that they preserve the CTL* properties. Section 5 explains how to encode variational systems using high-level modelling languages and presents how abstractions are implemented as source-to-source transformations of such high-level models. The evaluation on several case studies is presented in Sect. 6. In Sect. 7, we show how our verification procedure can be extended in order to (1) handle μ-calculus properties and (2) become a fully automatic procedure. Finally, we discuss the relation to other works and conclude.

2 Motivating example

A variability model for the VendingMachine [10] variational system is shown in Fig. 1. It describes the behaviours of a family of models of vending machines in an aggregating form using a featured transition system [10]. The VendingMachine family has five features, and each of them is assigned an identifying letter and a color. The features are: `VendingMachine` (denoted by v, in black), the mandatory base feature for purchasing a drink which is enabled in all variants of this family; `Tea` (t, in red), for serving tea; `Soda` (s, in green), for serving soda, which is also a mandatory feature present in all variants; `CancelPurchase` (c, in brown), for cancelling a purchase after a coin is entered; and `FreeDrinks` (f, in blue) for offering free drinks. Each transition is labelled by first an *action*, then a slash '/', and a guarding *feature expression* specifying for which variants the transition is to be included (this depends on which features have been enabled in a given variant). For example, the transition ① $\xrightarrow{free/f}$ ③ is included in variants that have the feature f enabled.

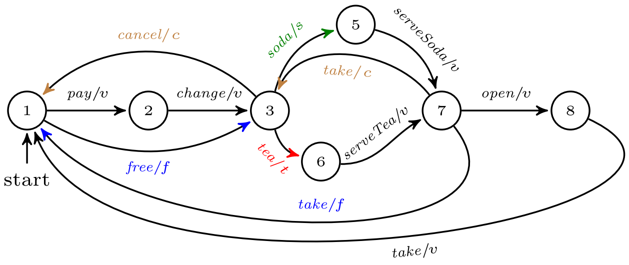
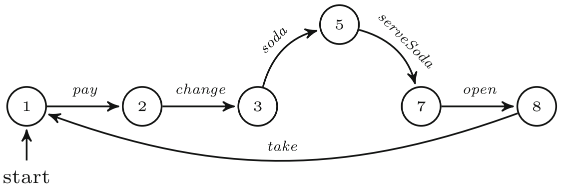
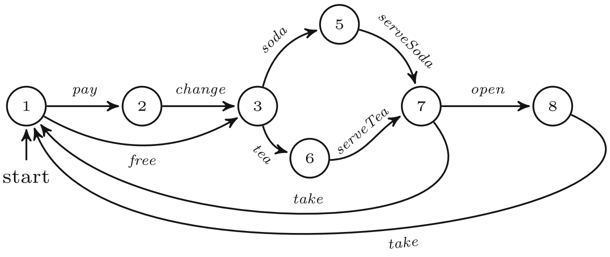


Fig. 1 The variability model for VendingMachine



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



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

Fig. 2 Some variants of VendingMachine

By combining various features, a number of variants of the VendingMachine family can be derived. Figure 2a shows a basic variant of VendingMachine that only serves soda. Only the features v and s are enabled, so it is described by the configuration $\{v, s\}$. The machine takes a coin, returns change, serves a soda, opens the access compartment so that the customer can take the soda, before closing it again. Figure 2b shows another variant of this machine (features v, s, t, and f enabled), which serves both tea and soda, and offers free drinks as well as paid drinks. Other variants can be derived by enabling other combinations of features. We can derive up to 2^n variants, where n is the number of features available in the family. In general, not all combinations of features give rise to *valid* variants (configurations). For the VendingMachine family, the features v and s are mandatory, so they must be enabled in all valid variants. The other features t, c, and f are optional, thus yielding eight valid variants in this family.

Suppose that a proposition `start` holds in the initial state ①. Consider the following property

> Φ_1: *in every state along every execution, there exists a possible continuation that will eventually reach the* `start` *state.*

Both variants of VendingMachine in Fig. 2 satisfy this property, since the initial state ① is reachable from any state of those variants. In fact, all variants of VendingMachine satisfy Φ_1.

Model checking [3] can be used to verify formally whether properties like Φ_1 hold for the VendingMachine family. One possibility is to instantiate all valid variants of the family and verify them one by one, using a standard single-system model checker (this is known as "brute-force" approach). Alternatively, we can use a specialized family-based model checking algorithms and tools [11] that operate directly on the variability model. Although the family-based model checkers are more efficient than the brute-force approach, their efficiency still depends on the number of features and variants in the family. Moreover, family-based model checkers require a costly design and implementation.

In order to overcome the above problems of family-based approach (scalability and development cost), we previously proposed to use *variability abstractions* [17,18]. They reduce the number of variants, producing abstract variability models which are smaller than the concrete ones. Hence, the model checking of these models is computationally more efficient, but less precise. However, the variability abstractions introduced in [17,18] are conservative, which means that the resulting abstract variability models over-approximate the concrete ones and are sound only with respect to universal properties (that contain just the universal ∀ path quantifier.) On the other hand, the above property Φ_1 is not universal, since it contains both universal quantifiers (for all executions, ∀) and existential quantifiers (there exists an execution, ∃). In order to handle such properties, we have to use both conservative (over-approximating) variability abstractions and their dual (under-approximating) variability abstractions. In effect, we will obtain an abstract variability model (known as modal transition system), which has two types of transitions: *may*-transitions defined using the conservative abstractions, and *must*-transitions defined using the dual abstractions. The property Φ_1 is now interpreted over abstract variability models as follows:

> Φ_1: *in every state along every may-execution (that contains only may-transitions), there exists a possible must-continuation (that contains only must-transitions) that will eventually reach the* `start` *state.*

If the property holds in such an abstract variability model, then it will hold in the concrete variability model as well (by our soundness result).

We consider a basic variability abstraction, called the *join abstraction* (written $\boldsymbol{\alpha}^{\text{join}}$). It merges all valid variants into a single abstract model, such that its may-part contains all transitions that occur in at least one variant, whereas its must-part contains only those transitions that occur in all variants. Note that with $\boldsymbol{\alpha}^{\text{join}}$ we obtain a single-system model with no variability in it. We combine the simplification of concrete models by the variability abstraction with a divide-and-conquer strategy for more efficient verification. The key operator for this strategy is *projection* (denoted π), which can be used to partition the configuration space of all variants into disjoint subsets, producing several concrete variability models that can be analysed (and abstracted) separately.

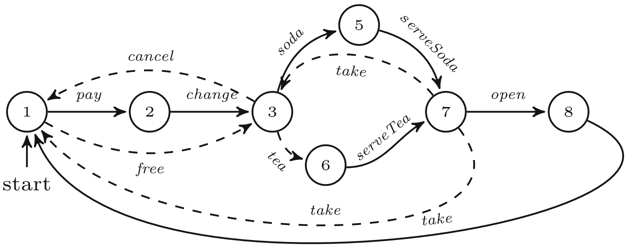


Fig. 3 The abstract model $\boldsymbol{\alpha}^{\text{join}}$(VendingMachine). Dashed edges represent may-transitions, while solid edges represent must-transitions

We illustrate the use of abstractions via our example property Φ_1. We apply the $\boldsymbol{\alpha}^{\text{join}}$ abstraction on the variability model of Fig. 1, which simply joins control flows of all variants into a single model (known as modal transition system), where all (mandatory and optional) features become *true* in may-transitions and only mandatory features become *true* in must-transitions (the other optional features become *false*). As a result of this operation, we obtain the abstract model $\boldsymbol{\alpha}^{\text{join}}$(VendingMachine), shown in Fig. 3, where may-transitions are denoted by dashed lines and must-transitions are denoted by solid lines. Note that every must-transition is also a may-transition, and we only show those (may and must) transitions whose presence conditions have evaluated to *true* after applying the join abstraction. The may-part of $\boldsymbol{\alpha}^{\text{join}}$(VendingMachine), which contains only may-executions, over-approximates the VendingMachine in the sense that it contains more executions than VendingMachine. On the other hand, the must-part of $\boldsymbol{\alpha}^{\text{join}}$(VendingMachine), which contains only must-executions, under-approximates the VendingMachine in the sense that it contains less executions than VendingMachine. The variability-specific information about features is lost in the abstract model. The may- and must-parts of $\boldsymbol{\alpha}^{\text{join}}$(VendingMachine) represent ordinary transition systems, and they can be verified efficiently using standard (single-system) model checkers (such as NuSMV). It can be shown that an abstract variability model satisfies one property, if both its may- and must-parts satisfy the same property. Thus, by verifying that Φ_1 holds for the may- and must-parts of $\boldsymbol{\alpha}^{\text{join}}$(VendingMachine), we have that Φ_1 holds for $\boldsymbol{\alpha}^{\text{join}}$(VendingMachine). Then, using the soundness result for the join abstraction we can conclude that Φ_1 holds for the concrete VendingMachine as well (i.e. for all its valid variants).

3 Background

We begin by summarizing the existing background for our work. We define modelling formalisms for describing single systems and then proceed with modelling formalisms for variational systems. Finally, we present the temporal logic CTL*, which is used to specify system properties.

3.1 Single systems

We present the basic definition of a transition system (TS) and a modal transition system (MTS) that we will use to describe behaviours of single systems.

Definition 1 A transition system (TS) is a tuple $\mathcal{T} = (S, Act, trans, I, AP, L)$, where S is a finite set of states; Act is a finite set of actions; $trans \subseteq S \times Act \times S$ is a transition relation which is *total*, so that for each state there is an outgoing transition; $I \subseteq S$ 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 $[\![\mathcal{T}]\!]_{\text{TS}}$, is the set of its executions.

Remark TSs are used for modelling reactive systems whose computations typically do not terminate. In such systems, terminal states in which no progress is possible are undesirable and often represent a design error. Therefore, we consider TSs without terminal states (transition relation is total) and only infinite sequences.

MTSs [35] are a generalization of transition systems that allows describing not just a sum of all behaviours of a system but also an over- and under-approximation of the system's behaviours. An MTS is a TS equipped with two transition relations: *must* and *may*. The former (must) is used to specify the required behaviour, while the latter (may) to specify the allowed behaviour of a system. We will use MTSs for representing abstractions of variational systems.

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

The intuition behind the inclusion $trans^{\text{must}} \subseteq trans^{\text{may}}$ is that transitions that are necessarily true ($trans^{\text{must}}$) are also possibly true ($trans^{\text{may}}$). A *may-execution* in $\mathcal{M}$ is an execution (infinite sequence) with all its transitions in $trans^{\text{may}}$, whereas a *must-execution* in $\mathcal{M}$ is a maximal sequence with all its transitions in $trans^{\text{must}}$, which cannot be extended with any other transition from $trans^{\text{must}}$. Note that since $trans^{\text{must}}$ is not necessarily total, must-executions can be finite. We use $[\![\mathcal{M}]\!]^{\text{may}}_{\text{MTS}}$ to denote the set of all may-executions in $\mathcal{M}$, whereas $[\![\mathcal{M}]\!]^{\text{must}}_{\text{MTS}}$ to denote the set of all must-executions in $\mathcal{M}$.

3.2 Variational 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: $k(A_1) \wedge \cdots \wedge k(A_n)$, 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.

An FTS describes behaviour 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, $FeatExp(\mathbb{F})$, which are propositional logic formulae over $\mathbb{F}$:

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

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 $k \models \psi$, where $\models$ is the standard satisfaction relation of propositional logic. For example, given $\mathbb{F} = \{A, B\}$ with all four possible variants being valid, we get: $[\![A \vee B]\!] = \{A \wedge B, A \wedge \neg B, \neg A \wedge B\}$.

Definition 3 A 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 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 $(S, Act, trans', I, AP, L)$, where $trans' = \{t \in trans \mid k \models \delta(t)\}$. 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 $(S, Act, trans', I, AP, L, \mathbb{F}, \mathbb{K}', \delta')$, where $trans' = \{t \in trans \mid \exists k \in \mathbb{K}'. k \models \delta(t)\}$ and $\delta' = \delta|_{trans'}$ is the restriction of δ to $trans'$. The *semantics* of an FTS $\mathcal{F}$, denoted as

$[\![\mathcal{F}]\!]_{\mathrm{FTS}}$, is the union of behaviours of the projections on all valid variants $k \in \mathbb{K}$, i.e. $[\![\mathcal{F}]\!]_{\mathrm{FTS}} = \cup_{k\in\mathbb{K}}[\![\pi_k(\mathcal{F})]\!]_{\mathrm{TS}}$.

Example 1 Consider the FTS for the VendingMachine family presented in Fig. 1. The FTS has five features $\mathbb{F} = \{v, t, s, c, f\}$. The set of all valid configurations is obtained by combining the above features. Recall that v and s are mandatory features, so the set of valid configurations is: $\mathbb{K}^{\mathrm{VM}} = \{\{v, s\}, \{v, s, t\}, \{v, s, c\}, \{v, s, t, c\}, \{v, s, f\}, \{v, s, t, f\}, \{v, s, c, f\}, \{v, s, t, c, f\}\}$.

Figure 2a shows a basic version of VendingMachine that only serves soda. This variant is described by the configuration: $\{v, s\}$, equivalently $v \wedge s \wedge \neg t \wedge \neg c \wedge \neg f$. The model presented in the figure is obtained by the projection $\pi_{\{v,s\}}$(VendingMachine). Similarly, we can obtain the model $\pi_{\{v,s,t,f\}}$(VendingMachine) shown in Fig. 2b.

Figure 3 shows an MTS, where must-transitions are denoted by solid lines and may-transitions are denoted by dashed lines. □

3.3 CTL* properties

Computation tree logic* (CTL*) [3,7] 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:

$$\Phi ::= true \mid false \mid a \mid \neg a \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid \forall\phi \mid \exists\phi,$$
$$\phi ::= \Phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \bigcirc\phi \mid \phi_1 \mathsf{U} \phi_2 \mid \phi_1 \mathsf{V} \phi_2$$

where $a \in AP$ and ϕ represent CTL* path formulae. The path formula $\bigcirc\phi$ can be read as "from the next state ϕ", while $\phi_1 \mathsf{U} \phi_2$ can be read as "ϕ_1 until ϕ_2", and $\phi_1 \mathsf{V} \phi_2$ can be read as "ϕ_2 while not ϕ_1" (where ϕ_1 may never hold). Other derived temporal operators for path formulae can be defined as well by means of syntactic sugar, for instance: $\Diamond\phi = true\,\mathsf{U}\phi$ (ϕ holds eventually), and $\Box\phi = \neg\Diamond\neg\phi$ (ϕ always holds). Note that the CTL* state formulae Φ are given in negation normal form ($\neg$ is applied only to atomic propositions). This facilitates the definition of $\forall$CTL* and $\exists$CTL*, which are subsets of CTL* where the only allowed path quantifiers are $\forall$ and $\exists$, respectively. Given $\Phi \in$ CTL*, we consider $\neg\Phi$ to be the equivalent CTL* formula given in negation normal form. To ensure that every CTL* formula is equivalent to a formula in negation normal form, for each operator the corresponding dual operator is necessary. We have that $\wedge$ and $\vee$ are dual, $\bigcirc$ is dual to itself, as well as U and V are dual. For example, we use the duality law: $\neg\forall\bigcirc\Phi \equiv \exists\bigcirc\neg\Phi$.

We formalize 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 ith 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 ith state.

Definition 4 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 $s \models \Phi_2$;
$s \models \Phi_1 \vee \Phi_2$ iff $s \models \Phi_1$ or $s \models \Phi_2$
(3) $s \models \forall\phi$ iff $\forall\rho \in [\![\mathcal{T}]\!]^s_{\mathrm{TS}}.\, \rho \models \phi$;
$s \models \exists\phi$ iff $\exists\rho \in [\![\mathcal{T}]\!]^s_{\mathrm{TS}}.\, \rho \models \phi$

Satisfaction of a path formula ϕ for an execution $\rho = s_0\lambda_1 s_1\ldots$ 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 $\rho \models \phi_2$;
$\rho \models \phi_1 \vee \phi_2$ iff $\rho \models \phi_1$ or $\rho \models \phi_2$;
$\rho \models \bigcirc\phi$ iff $\rho_1 \models \phi$;
$\rho \models (\phi_1 \mathsf{U} \phi_2)$ iff $\exists i \geq 0.\, \big(\rho_i \models \phi_2 \wedge (\forall 0 \leq j < i.\, \rho_j \models \phi_1)\big)$
$\rho \models (\phi_1 \mathsf{V} \phi_2)$ iff $\forall i \geq 0.\, \big(\forall 0 \leq j < i.\, \rho_j \not\models \phi_1 \implies \rho_i \models \phi_2\big)$

A TS $\mathcal{T}$ satisfies a state formula Φ, written $\mathcal{T} \models \Phi$, iff all its initial states satisfy the formula: $\forall s_0 \in I.\, s_0 \models \Phi$.

We say that an FTS $\mathcal{F}$ *satisfies a* CTL* *formula* Φ, written $\mathcal{F} \models \Phi$, iff all its valid variants satisfy the formula: $\forall k \in \mathbb{K}.\, \pi_k(\mathcal{F}) \models \Phi$. Otherwise, we say that $\mathcal{F}$ does not satisfy Φ, written $\mathcal{F} \not\models \Phi$. In this case, we also want to determine a non-empty set of violating variants $\mathbb{K}' \subseteq \mathbb{K}$, such that $\forall k' \in \mathbb{K}'.\, \pi_{k'}(\mathcal{F}) \not\models \Phi$ and $\forall k \in \mathbb{K}\backslash\mathbb{K}'.\, \pi_k(\mathcal{F}) \models \Phi$.

We now define the semantics of CTL* over an MTS $\mathcal{M}$. We define $\mathcal{M}, s \models \Phi$ and $\mathcal{M}, \rho \models \phi$, which are slightly different from Definition 4 where a TS $\mathcal{T}$ is considered. In particular, the clause (3) is replaced by:

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

An MTS $\mathcal{M}$ satisfies a state formula Φ, written $\mathcal{M} \models \Phi$, iff $\forall s_0 \in I.\, s_0 \models \Phi$.

Note that the duality laws no longer hold for MTSs. However, we use MTSs as abstract variability models that only help to speed up the procedure for verifying CTL* properties over FTSs (i.e. a family of TSs).

Example 2 Consider the FTS VendingMachine in Fig. 1. We restate the example property Φ_1 from Sect. 2 in CTL* as follows:

$$\Phi_1 = \forall\Box\,\exists\Diamond\,\mathtt{start}$$

where the proposition `start` holds in the initial state ①. The property states that 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 $\textsc{VendingMachine} \models \Phi_1$, since Φ_1 holds for all variants. Even for variants with the feature c enabled, there is a continuation from the state ③ back to ①.

Consider another property

$$\Phi_2 = \forall\Box\,\forall\Diamond\texttt{start}$$

It 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 $\textsc{VendingMachine} \not\models \Phi_2$. 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, note that the variants from $[\![\neg c]\!]$ do satisfy Φ_2, that is, $\pi_{[\![\neg c]\!]}(\textsc{VendingMachine}) \models \Phi_2$.

Finally, consider the $\exists$CTL* property

$$\Phi_3 = \exists\Box\,\exists\Diamond\texttt{start}$$

It 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 witness is ① → ② → ③ → ⑤ → ⑦ → ⑧ → ① ..., which is an execution that belongs to all valid variants. □

CTL* extends CTL as it allows path quantifiers $\forall$ and $\exists$ to be arbitrarily nested with temporal operators, such as $\bigcirc$ and U. In contrast, in CTL each temporal operator must be immediately preceded by a path quantifier. Hence, all properties in Example 2 are from CTL. We now show how to handle a CTL* property which is not in CTL.

Example 3 Let the proposition `selectedTea` holds only in the state ⑥, and it does not hold in all other states. We want to check the property:

$$\Phi_4 = \forall\Diamond\Box(\neg\texttt{selectedTea})$$

It states that along every execution there exists a state such that after that state `selectedTea` does not hold forever. Note that $\textsc{VendingMachine} \not\models \Phi_4$. For example, if the feature t (`Tea`) is enabled, a possible counter-example is: ① → ③ → ⑥ → ⑦ → ① → .... The set of violating products is $[\![t]\!]$. However, note that the variants from $[\![\neg t]\!]$ do satisfy Φ_4, that is, $\pi_{[\![\neg t]\!]}(\textsc{VendingMachine}) \models \Phi_4$. □

4 Abstraction of variational systems

We now introduce the variability abstractions which preserve full CTL*, including its universal and existential properties. The abstractions simplify the configuration space of a variational system (i.e. the corresponding FTS), by reducing the number of configurations and manipulating presence conditions of transitions. We start working with variability abstractions between Boolean complete lattices of feature expressions and then induce a notion of abstract models of FTSs. Finally, we show that the obtained abstract models are sound with respect to CTL* properties, and illustrate how those abstract models can be combined with a divide-and-conquer verification strategy based on partitioning the configuration space.

4.1 Variability abstractions

An *abstraction* is a mapping of elements in a concrete domain (say concrete models) to elements of an abstract domain (here the abstract models). Usually, we want to simplify representation of the object under analysis to speed up analysis. In program analysis and verification, domains are usually complete lattices, which have sufficiently rich structure that facilitates ordering of elements (some elements are more informative than others) and synthesizing new elements using least upper bound.

Let $\langle L, \leqslant_L\rangle$ and $\langle M, \leqslant_M\rangle$ be complete lattices taking the role of the *concrete* and *abstract* domain, respectively. A *Galois connection* is a pair of total functions, $\alpha : L \to M$ and $\gamma : M \to L$ (respectively, known as the *abstraction* and *concretization* functions) such that:

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

We write $\langle L, \leqslant_L\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle M, \leqslant_M\rangle$ to state that (α, γ) are a Galois connection between L and M. We will use Galois connections to approximate a computationally expensive (or uncomputable) analysis (model) formulated over L with a computationally cheaper analysis (model) formulated over M.

Variability abstractions simplify the configuration space of an FTS, by reducing the number of configurations. This is easy to do by manipulating presence conditions of transitions. Thus, we define abstraction of variability in FTSs primarily by abstraction of presence conditions, working with Galois connections between standard Boolean complete lattices. We define two classes of abstractions. We use the standard conservative abstractions [17,18] 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 defined as: $(FeatExp(\mathbb{F})_{/\equiv}, \models, \vee, \wedge, true, false, \neg)$. The elements of the domain $FeatExp(\mathbb{F})_{/\equiv}$ are equivalence classes of propositional formulae $\psi \in FeatExp(\mathbb{F})$ 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*, $\boldsymbol{\alpha}^{\text{join}}$, merges the control flow of all variants, obtaining a single variant that includes all executions that may occur 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: $\boldsymbol{\alpha}^{\text{join}}(\mathbb{F}) = \emptyset$, and the abstract set of valid configurations is a singleton: $\boldsymbol{\alpha}^{\text{join}}(\mathbb{K}) = \{true\}$ if $\mathbb{K} \neq \emptyset$. The abstraction and concretization functions between $FeatExp(\mathbb{F})$ and $FeatExp(\emptyset)$, forming a Galois connection [17,18], are defined as:

$$\boldsymbol{\alpha}^{\text{join}}(\psi) = \begin{cases} true & \text{if } \exists k \in \mathbb{K}. k \models \psi \\ false & \text{otherwise} \end{cases}$$
$$\boldsymbol{\gamma}^{\text{join}}(\psi) = \begin{cases} true & \text{if } \psi \text{ is } true \\ \bigvee_{k \in 2^{\mathbb{F}} \setminus \mathbb{K}} k & \text{if } \psi \text{ is } false \end{cases} \tag{1}$$

Dual abstractions. Suppose that $\langle FeatExp(\mathbb{F})_{/\equiv}, \models \rangle$, $\langle FeatExp(\alpha(\mathbb{F}))_{/\equiv}, \models \rangle$ are Boolean complete lattices, and $\langle FeatExp(\mathbb{F})_{/\equiv}, \models \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle FeatExp(\alpha(\mathbb{F}))_{/\equiv}, \models \rangle$ is a Galois connection. We define [15]: $\widetilde{\alpha} = \neg \circ \alpha \circ \neg$ and $\widetilde{\gamma} = \neg \circ \gamma \circ \neg$ so that $\langle FeatExp(\mathbb{F})_{/\equiv}, \models \rangle \underset{\widetilde{\alpha}}{\overset{\widetilde{\gamma}}{\leftrightarrows}} \langle FeatExp(\alpha(\mathbb{F}))_{/\equiv}, \models \rangle$ is a Galois connection (or equivalently, we have that $\langle FeatExp(\alpha(\mathbb{F}))_{/\equiv}, \models \rangle \underset{\widetilde{\gamma}}{\overset{\widetilde{\alpha}}{\leftrightarrows}} \langle FeatExp(\mathbb{F})_{/\equiv}, \models \rangle$). 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 $FeatExp(\mathbb{F})$ and $FeatExp(\emptyset)$, 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:

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

4.2 Abstract MTS

Given a Galois connection $(\boldsymbol{\alpha}^{\text{join}}, \boldsymbol{\gamma}^{\text{join}})$ defined on the level of feature expressions, we now define the abstraction of an FTS as an MTS 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 5 Let $\mathcal{F} = (S, Act, trans, I, AP, L, \mathbb{F}, \mathbb{K}, \delta)$ be an FTS, we define its abstraction to be the MTS $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F}) = (S, Act, trans^{\text{may}}, trans^{\text{must}}, I, AP, L)$, where $trans^{\text{may}} = \{t \in trans \mid \boldsymbol{\alpha}^{\text{join}}(\delta(t)) = true\}$, and also $trans^{\text{must}} = \{t \in trans \mid \widetilde{\boldsymbol{\alpha}^{\text{join}}}(\delta(t)) = true\}$.

Note that the abstract model $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F})$ has no variability in it, i.e. it contains only one abstract configuration (that is, $true \in \boldsymbol{\alpha}^{\text{join}}(\mathbb{K})$).

Example 4 Recall the FTS VendingMachine of Fig. 1 with the set of valid configurations $\mathbb{K}^{\text{VM}}$ (see Example 1). Figure 3 shows $\boldsymbol{\alpha}^{\text{join}}$(VendingMachine), where the allowed (may) part of the behaviour 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 $\boldsymbol{\alpha}^{\text{join}}$(VendingMachine) is an ordinary MTS with no variability. □

From the MTS $\mathcal{M}$, we define two TSs $\mathcal{M}^{\text{may}}$ and $\mathcal{M}^{\text{must}}$ representing the may- and must-components of $\mathcal{M}$, i.e. they only contain may- and must-transitions of $\mathcal{M}$, respectively. Thus, we have $[\![\mathcal{M}^{\text{may}}]\!]_{\text{TS}} = [\![\mathcal{M}]\!]_{\text{MTS}}^{\text{may}}$ and $[\![\mathcal{M}^{\text{must}}]\!]_{\text{TS}} = [\![\mathcal{M}]\!]_{\text{MTS}}^{\text{must}}$.

4.3 Preservation of CTL*

We now show that the abstraction of an FTS is sound with respect to CTL*. First, we show two helper lemmas stating that: if any variant $k \in \mathbb{K}$ that can execute a behaviour, then the abstract model $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F})$ can execute the same may-behaviour; and if the abstract model $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F})$ can execute a must-behaviour, then all valid variants $k \in \mathbb{K}$ can execute the same behaviour.

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, $\boldsymbol{\alpha}^{join}(\psi) = true$.*
(ii) *Let $\widetilde{\boldsymbol{\alpha}^{join}}(\psi) = true$. Then, for all $k \in \mathbb{K}$, it holds $k \models \psi$.*

Proof (i) By assumption, we have that $\exists k \in \mathbb{K}. k \models \psi$. Thus, by definition of $\boldsymbol{\alpha}^{\text{join}}$ in Eq. (1), we have $\boldsymbol{\alpha}^{\text{join}}(\psi) = true$.

(ii) By assumption, we have that $\widetilde{\boldsymbol{\alpha}^{\text{join}}}(\psi) = \mathit{true}$. By definition of $\widetilde{\boldsymbol{\alpha}^{\text{join}}}$ in Eq. (2), this is the case only if for all $k \in \mathbb{K}$, it holds $k \models \psi$. □

Lemma 2 (i) *Let $k \in \mathbb{K}$ and $\rho \in [\![\pi_k(\mathcal{F})]\!]_{TS}$. Then, $\rho \in [\![\boldsymbol{\alpha}^{join}(\mathcal{F})]\!]^{may}_{MTS}$.*

(ii) *Let $\rho \in [\![\boldsymbol{\alpha}^{join}(\mathcal{F})]\!]^{must}_{MTS}$. Then, $\rho \in [\![\pi_k(\mathcal{F})]\!]_{TS}$ for all $k \in \mathbb{K}$.*

Proof (i) Let $\rho = s_0\lambda_1 s_1\lambda_2 \ldots \in [\![\pi_k(\mathcal{F})]\!]_{\text{TS}}$ for some $k \in \mathbb{K}$. 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 $\boldsymbol{\alpha}^{\text{join}}(\delta(t_i)) = \mathit{true}$ for all $i \geq 0$. Hence, we have $t_i \in \mathit{trans}^{\text{may}}$ for $i \geq 0$, and so $\rho \in [\![\boldsymbol{\alpha}^{\text{join}}(\mathcal{F})]\!]^{\text{may}}_{\text{MTS}}$.

(ii) Let $\rho = s_0\lambda_1 s_1\lambda_2 \ldots \in [\![\boldsymbol{\alpha}^{\text{join}}(\mathcal{F})]\!]^{\text{must}}_{\text{MTS}}$. This means that for all transitions in ρ, $t_i = s_i \xrightarrow{\lambda_{i+1}} s_{i+1}$, we have that $t_i \in \mathit{trans}^{\text{must}}$ and so $\widetilde{\boldsymbol{\alpha}^{\text{join}}}(\delta(t_i)) = \mathit{true}$ for all $i \geq 0$. By Lemma 1(ii), we have that for all $k \in \mathbb{K}$, it holds $k \models \delta(t_i)$ for all $i \geq 0$. Hence, we have $\rho \in [\![\pi_k(\mathcal{F})]\!]_{\text{TS}}$ for all $k \in \mathbb{K}$. □

As a result, every ∀CTL⋆ (resp., ∃CTL⋆) property true for the may- (resp., must-) component of $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F})$ is true for $\mathcal{F}$ as well. Moreover, the MTS $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F})$ preserves the full CTL⋆.

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

(∀CTL⋆) *For any $\Phi \in \forall CTL^{\star}$, $\boldsymbol{\alpha}^{join}(\mathcal{F})^{may} \models \Phi \implies \mathcal{F} \models \Phi$.*

(∃CTL⋆) *For any $\Phi \in \exists CTL^{\star}$, $\boldsymbol{\alpha}^{join}(\mathcal{F})^{must} \models \Phi \implies \mathcal{F} \models \Phi$.*

(CTL⋆) *For any $\Phi \in CTL^{\star}$, $\boldsymbol{\alpha}^{join}(\mathcal{F}) \models \Phi \implies \mathcal{F} \models \Phi$.*

Proof We prove the most difficult case (CTL⋆).

By induction on the structure of Φ. We prove for state formulae Φ that if $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F}) \models \Phi$, then $\mathcal{F} \models \Phi$ (i.e. for all $k \in \mathbb{K}$, $\pi_k(\mathcal{F}) \models \Phi$). All cases except ∀ and ∃ quantifiers are straightforward.

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

For $\Phi = \exists\phi$. Assume $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F}) \models \exists\phi$. This means that there exists an execution $\rho \in [\![\boldsymbol{\alpha}^{\text{join}}(\mathcal{F})]\!]^{\text{must}}_{\text{MTS}}$ such that $\rho \models \phi$. By Lemma 2(ii), we have that for all $k \in \mathbb{K}$, we have $\rho \in [\![\pi_k(\mathcal{F})]\!]_{\text{TS}}$, and so $\pi_k(\mathcal{F}) \models \exists\phi$. Since $\pi_k(\mathcal{F}) \models \exists\phi$ for all $k \in \mathbb{K}$, it follows $\mathcal{F} \models \exists\phi$. □

The preservation results (soundness) from Theorem 1 mean that abstract models are designed to be conservative for the satisfaction of CTL⋆ properties. However, in case of the refutation of a property, the counter-example found in the abstract model may be spurious (introduced due to abstraction) for some variants and genuine for the others. This can be established by checking which concrete variants can execute the found counter-example.

Let Φ be a CTL⋆ formula which is not in ∀CTL⋆ nor in ∃CTL⋆, and let $\mathcal{M}$ be an MTS. We can verify $\mathcal{M} \models \Phi$ by checking Φ on two TSs $\mathcal{M}^{\text{may}}$ and $\mathcal{M}^{\text{must}}$ and then by combining the obtained results as specified below.

Theorem 2 *For any $\Phi \in CTL^{\star}$ and MTS $\mathcal{M}$, we have:*

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

Proof By induction on the structure of Φ. All cases except ∀ and ∃ quantifiers are straightforward.

For $\Phi = \forall\phi$. Consider the first case, when $\mathcal{M} \models \Phi = \mathit{true}$. Assume $\mathcal{M}^{\text{may}} \models \forall\phi$. That is, for any may-execution ρ of $\mathcal{M}$ we have $\rho \models \phi$. By Definition 4 (3'), we have $\mathcal{M} \models \Phi$. Consider the second case, when $\mathcal{M} \models \Phi = \mathit{false}$. Assume $\mathcal{M}^{\text{may}} \not\models \forall\phi$. That is, there exists a may-execution ρ of $\mathcal{M}$ such that $\rho \not\models \phi$. By Definition 4 (3'), we have $\mathcal{M} \not\models \Phi$. Assume $\mathcal{M}^{\text{must}} \not\models \forall\phi$. That is, there exists a must-execution ρ of $\mathcal{M}$ such that $\rho \not\models \phi$. But ρ is also a may-execution, so by Definition 4 (3'), we have $\mathcal{M} \not\models \Phi$.

For $\Phi = \exists\phi$. Consider the first case, when $\mathcal{M} \models \Phi = \mathit{true}$. Assume $\mathcal{M}^{\text{must}} \models \exists\phi$. That is, there exists a must-execution ρ of $\mathcal{M}$ such that $\rho \models \phi$. By Definition 4 (3'), we have $\mathcal{M} \models \Phi$. Consider the second case, when $\mathcal{M} \models \Phi = \mathit{false}$. Assume $\mathcal{M}^{\text{may}} \not\models \exists\phi$. That is, for all may-executions ρ of $\mathcal{M}$ 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 4 (3'), we have $\mathcal{M} \not\models \Phi$. Assume $\mathcal{M}^{\text{must}} \not\models \exists\phi$. That is, for all must-executions ρ of $\mathcal{M}$ we have $\rho \not\models \phi$. By Definition 4 (3'), we have $\mathcal{M} \not\models \Phi$. □

Therefore, we can check whether the abstract model $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F})$ satisfies a formula Φ, which is not in ∀CTL⋆ nor in ∃CTL⋆, by running a model checker twice, once with the may-component of $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F})$ and once with the must-component of $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F})$. On the other hand, a formula Φ from ∀CTL⋆ (resp., ∃CTL⋆) is checked against $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F})$ by running a model checker only once with the may-component (resp., must-component) of $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F})$.

Divide-and-conquer strategy The family-based model checking problem $\mathcal{F} \models \Phi$ can be reduced to a number of smaller problems by partitioning the configuration space $\mathbb{K}$. 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.

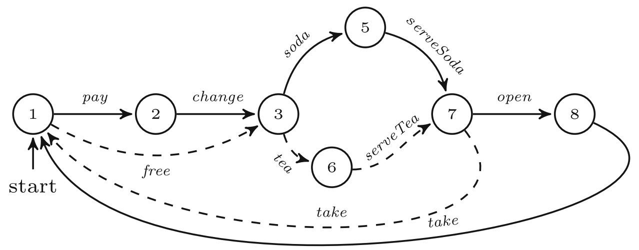


Fig. 4 $\alpha^{\mathrm{join}}(\pi_{[\![\neg c]\!]}(\textsc{VendingMachine}))$

Corollary 1 *Let* $\mathbb{K}_1, \mathbb{K}_2, \ldots, \mathbb{K}_n$ *form a* partition *of* $\mathbb{K}$. *If* $\alpha^{join}(\pi_{\mathbb{K}_1}(\mathcal{F})) \models \Phi, \ldots, \alpha^{join}(\pi_{\mathbb{K}_n}(\mathcal{F})) \models \Phi$, *then* $\mathcal{F} \models \Phi$.

Proof Assume $\alpha^{\mathrm{join}}(\pi_{\mathbb{K}_1}(\mathcal{F})) \models \Phi, \ldots, \alpha^{\mathrm{join}}(\pi_{\mathbb{K}_n}(\mathcal{F})) \models \Phi$. By Theorem 1 (CTL*), it follows that $\pi_{\mathbb{K}_1}(\mathcal{F}) \models \Phi, \ldots, \pi_{\mathbb{K}_n}(\mathcal{F}) \models \Phi$. Since $\mathbb{K}_1, \mathbb{K}_2, \ldots, \mathbb{K}_n$ form a partition of $\mathbb{K}$, we have that $\pi_k(\mathcal{F}) \models \Phi$ for all $k \in \mathbb{K}$. Hence, $\mathcal{F} \models \Phi$. □

Therefore, in case of suitable partitioning of $\mathbb{K}$ and the aggressive α^{join} abstraction, all $\alpha^{\mathrm{join}}(\pi_{\mathbb{K}_i}(\mathcal{F}))^{\mathrm{may}}$ and $\alpha^{\mathrm{join}}(\pi_{\mathbb{K}_i}(\mathcal{F}))^{\mathrm{must}}$ 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 5 Consider the properties introduced in Example 2. We can verify $\Phi_1 = \forall\Box\exists\Diamond\mathtt{start}$ by checking may- and must-components of $\alpha^{\mathrm{join}}(\textsc{VendingMachine})$. In particular, we have $\alpha^{\mathrm{join}}(\textsc{VendingMachine})^{\mathrm{may}} \models \Phi_1$ and $\alpha^{\mathrm{join}}(\textsc{VendingMachine})^{\mathrm{must}} \models \Phi_1$. Thus, using Theorem 1, (CTL*) and Theorem 2, we have that $\textsc{VendingMachine} \models \Phi_1$.

Using the TS $\alpha^{\mathrm{join}}(\textsc{VendingMachine})^{\mathrm{may}}$, we can verify $\Phi_2 = \forall\Box\forall\Diamond\mathtt{start}$ (Theorem 1, ($\forall$CTL*)). In this case, we obtain the counter-example ① → ③ → ⑤ → ⑦ → ③..., which is genuine for variants satisfying c. Hence, variants from $[\![c]\!]$ violate Φ_2. On the other hand, we can establish that variants from $[\![\neg c]\!]$ satisfy Φ_2 in the following way. First, we can construct the model $\alpha^{\mathrm{join}}(\pi_{[\![\neg c]\!]}(\textsc{VendingMachine}))$, which is shown in Fig. 4. Since $\alpha^{\mathrm{join}}(\pi_{[\![\neg c]\!]}(\textsc{VendingMachine}))^{\mathrm{may}}$ satisfies Φ_2, we can conclude by Theorem 1, ($\forall$CTL*) that all variants from $[\![\neg c]\!]$ satisfy Φ_2.

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

Example 6 Consider the property Φ_4 from Example 3. We can verify $\Phi_4 = \forall\Diamond\Box(\neg\mathtt{selectedTea})$ using the TS $\alpha^{\mathrm{join}}(\textsc{VendingMachine})^{\mathrm{may}}$ (Theorem 1, ($\forall$CTL*)). We obtain the counter-example ① → ③ → ⑥ → ⑦ → ① ..., which is genuine for variants satisfying t. Hence, variants from $[\![t]\!]$ violate Φ_4. But, we can check that $\alpha^{\mathrm{join}}(\pi_{[\![\neg t]\!]}(\textsc{VendingMachine}))^{\mathrm{may}} \models \Phi_4$, and thus by Theorem 1, ($\forall$CTL*) it follows that all variants from $[\![\neg t]\!]$ satisfy Φ_4. □

5 Implementation

We now describe an implementation of our abstraction-based approach for CTL model checking of variational systems in the context of state-of-the-art NuSMV model checker [6]. FTSs do not represent a convenient formalism for modelling very large variational systems. Hence, it is much appreciated by engineers to use some high-level modelling languages for FTSs. We use a high-level modelling language, called fNuSMV, which is expressively equivalent to FTSs and close to NuSMV's input language. First, we introduce the core NuSMV language, and then we present its feature-aware extension, fNuSMV. Finally, we show how to implement projections and variability abstractions as syntactic source-to-source transformations of high-level models specified in fNuSMV.

5.1 NuSMV language

The NuSMV modelling language [6] represents a high-level syntax for describing finite state automata. A NuSMV model consists of list of modules with parameters, which can be used to encapsulate and factor out recurring sub-models. A module (`MODULE`) contains variable declarations (`VAR`), macrodefinitions (`DEFINE`), assignments (`ASSIGN`), and properties (`SPEC`) to be checked. The variable declarations define the state space and the assignments define the transition relation of the finite state automaton described by the given model. Possible types for variables are Booleans, finite ranges of integers, and enumerations. The assignments are of the form:

$$\begin{array}{lll} s(v) := \mathtt{case} & b_1 : e_1; & \\ & \ldots & \\ & b_n : e_n; & \\ \quad \mathtt{esac} & & \end{array} \tag{3}$$

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, $\mathtt{init}(v)$, or $\mathtt{next}(v)$. We use v to assign the current value of v, $\mathtt{init}(v)$ to assign the initial value of v, and $\mathtt{next}(v)$ to assign the value of v in the next state. The next state value of v is given as a function of the variable values in the current state. The `case` statement is evaluated top to bottom, so the result is the expression from the first branch e_i whose condition b_i evaluates to *true*. Note that all assignments in a model are evaluated in parallel.

```
MODULE main
  VAR x : 0..m;
  ASSIGN
    init(x) := 0;
    next(x) := 0;
  SPEC AF(x ≥ k);
  SPEC AG(EF(x = k));
  SPEC EG(x ≥ k);
```

Fig. 5 A basic NuSMVmodel

Consider a basic NuSMV model shown in Fig. 5, with module name `main`. It consists of a single variable x which is initialized to 0 and does not change its value. This model defines an automaton with m states $(x = 0)$, …, $(x = m)$, where m is a meta-variable whose value depends on the number of features that will be composed with this basic model. The initial state is $(x = 0)$, and there is only one transition going from the initial state to itself. The CTL properties are "$\Phi_1 = \forall\Diamond(x \geq k)$", "$\Phi_2 = \forall\Box(\exists\Diamond(x = k))$", and "$\Phi_3 = \exists\Box(x \geq k)$", where k is a meta-variable that can be replaced with various natural numbers. For this model, all three properties hold when $k = 0$. In all other cases (for $k > 0$), the properties are violated.

5.2 Features in NuSMV

fNuSMV is a feature-oriented extension of NuSMV, which was introduced by Plath and Ryan [38] and subsequently improved by Classen [8]. It was shown in [8] that fNuSMV and FTS are expressively equivalent formalisms. The language 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. There are two main sections in the `FEATURE` construct, introduced by keywords `INTRODUCE` and `CHANGE`. In the `INTRODUCE` section, we place new variables that will be introduced in the basic model by composing the given feature with it. In the `CHANGE` section, we specify how the original state variables from the basic model are changed by using `TREAT` and `IMPOSE` clauses (they can be guarded by a condition). `TREAT` clauses can change the values of original variables when they are read, whereas `IMPOSE` clauses can override the definition of original variables.

For example, Fig. 6 shows a `FEATURE` construct, called A_1, which changes the basic model in Fig. 5. In particular, the feature A_1 defines a new Boolean variable nA_1, which is non-deterministically initialized. The basic system is changed in such a way that when nA_1 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_1 is set to *FALSE*. Otherwise, when nA_1 does not hold, the basic system is not changed.

```
FEATURE A1
  INTRODUCE
    VAR nA1 : boolean;
    ASSIGN init(nA1) := {TRUE,FALSE};
  CHANGE
    IF (nA1) THEN
    IMPOSE
      next(x) := x + 1 mod (m + 1);
      next(nA1) := next(x)=x+1?FALSE:nA1;
```

Fig. 6 The feature A_1

Classen [8] 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. Given a basic NuSMV model and a feature construct, the composed model is obtained as follows. Each feature A becomes a Boolean state variable fA, which is non-deterministically initialized and whose value never changes by the transitions. Thus, the initial states of the composed model include all possible feature combinations. Every change performed by a feature in the composition is guarded by the corresponding feature variable. Each declared variable or assignment from the `INTRODUCE` section is just added to the basic model. If in the `CHANGE` section we have a clause of the form

$$\texttt{IF}\ (b)\ \texttt{THEN}\ \texttt{IMPOSE}\ \ s(v) := e;$$

then we replace the assignments $s(v) := e'$ in the basic model with

$$s(v) := \texttt{case}\ \ \begin{array}{l} fA \,\&\, b : e; \\ \mathit{TRUE} : e'; \end{array} \qquad \texttt{esac} \tag{4}$$

Hence, when the feature A is enabled and the condition b is *true*, $s(v)$ is assigned the value of e, otherwise $s(v)$ is assigned the old value e' from the original basic model. In case we have the clause `IF` (b) `THEN TREAT` $v := e$, then all right-hand side occurrences of v (i.e. when v is read) in assignments of the basic model are replaced with the expression e when A is enabled and b holds. When several features are composed one after another, we assume that the features composed later take precedence over features composed earlier.

For example, the composition of the basic model and the feature A_1 given in Figs. 5 and 6 results in the model M_1 shown in Fig. 7. Note that, since the basic model is com-

```
MODULE features
  VAR fA1 : boolean;
  ASSIGN
    init(fA1) := {TRUE,FALSE};
    next(fA1) := fA1;
MODULE main
  VAR f : features;
      x : 0..1;
      nA1 : boolean;
   ASSIGN
     init(x) := 0;
     init(nA1) := {TRUE,FALSE};
     next(x) := case f.fA1 & nA1 : x+1 mod 2;
                          TRUE : 0;
                    easc;
     next(nA1) := case
                  f.fA1 & nA1 : next(x)=x+1?FALSE : nA1;
                  TRUE : nA1;
                       easc;
  SPEC AF(x ≥ 0);
  SPEC AG(EF(x = 0));
  SPEC EG(x ≥ 0);
```

Fig. 7 The composed model M_1

posed with one feature, the meta-variable m is instantiated to 1. First, a module, called `features`, containing all features (in this case, the single one A_1) is added to the system. To each feature (e.g. A_1) corresponds one variable in this module (e.g. fA_1). The variable fA_1 is non-deterministically initialized to *TRUE* or *FALSE*, and its value never changes in next states. The `main` module contains a variable named f of type `features`, so that all feature variables can be referenced in it (e.g. $f.fA_1$). In the next state, the variable x is incremented by 1 when the feature A_1 is enabled (fA_1 is *TRUE*) and nA_1 holds. Otherwise (*TRUE:* can be read as *else:*), x becomes zero. Also, nA_1 is set to *FALSE* when A_1 is enabled, nA_1 holds, and x is incremented by 1. Otherwise, nA_1 is not changed. The properties Φ_1, Φ_2, and Φ_3 with $k = 0$ hold for both variants when A_1 is enabled (fA_1 is *TRUE*) and A_1 is disabled (fA_1 is *FALSE*).

5.3 Syntactic transformations

We now show that projections and variability abstractions can be implemented as simple syntactic source-to-source transformations of fNuSMV models. This means that we do not need to build and store in memory the concrete full-blown FTS before applying an abstraction or projection to it, but we can effectively compute the abstract or projected model syntactically from the high-level modelling language. More specifically, let M be an fNuSMV model with a set of features $\mathbb{F}$ and a set of configurations $\mathbb{K}$, and let $[\![M]\!]$ denote the FTS obtained by its compilation. We can define $\boldsymbol{\alpha}^{\text{join}}(M)^{\text{may}}$ and $\boldsymbol{\alpha}^{\text{join}}(M)^{\text{must}}$ as syntactic transformations, such that $\boldsymbol{\alpha}^{\text{join}}([\![M]\!])^{\text{may}}_{\text{MTS}} = [\![\boldsymbol{\alpha}^{\text{join}}(M)^{\text{may}}]\!]$ and $\boldsymbol{\alpha}^{\text{join}}([\![M]\!])^{\text{must}}_{\text{MTS}} = [\![\boldsymbol{\alpha}^{\text{join}}(M)^{\text{must}}]\!]$. In other words, the may- (resp., must-) component of the abstract model obtained by applying $\boldsymbol{\alpha}^{\text{join}}$ on the FTS $[\![M]\!]$, that is, $\boldsymbol{\alpha}^{\text{join}}([\![M]\!])^{\text{may}}_{\text{MTS}}$ (resp., $\boldsymbol{\alpha}^{\text{join}}([\![M]\!])^{\text{must}}_{\text{MTS}}$), coincides with the TS obtained by compiling the NuSMV model $\boldsymbol{\alpha}^{\text{join}}(M)^{\text{may}}$ (resp., $\boldsymbol{\alpha}^{\text{join}}(M)^{\text{must}}$), that is, the TS $[\![\boldsymbol{\alpha}^{\text{join}}(M)^{\text{may}}]\!]$ (resp., $[\![\boldsymbol{\alpha}^{\text{join}}(M)^{\text{must}}]\!]$). The same applies for projections $\pi_{[\![\psi]\!]}$.

We now describe the rewrites for projection and join abstraction in detail. 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 using the `INVAR` construct of NuSMV to add the feature expression ψ' as an invariant to the model M. Another solution is to add the feature expression ψ' to each condition b_i in the assignments [which are of the form in Eq. (3)] to the state variables.

The abstracts $\boldsymbol{\alpha}^{\text{join}}(M)^{\text{may}}$ and $\boldsymbol{\alpha}^{\text{join}}(M)^{\text{must}}$ are obtained as follows. Let $s(v) := rhs$ be an assignment in the composed model M of the form in Eq. (4), obtained from changes made by a feature A to a basic model. In $\boldsymbol{\alpha}^{\text{join}}(M)^{\text{may}}$, if $\boldsymbol{\alpha}^{\text{join}}(A) = \mathit{true}$ and $\boldsymbol{\alpha}^{\text{join}}(\neg A) = \mathit{true}$ (that is, A is an optional feature), the above assignment becomes:

$$s(v) := \texttt{case}\ \ b : \{e, e'\}; \mathit{TRUE} : e';\ \ \texttt{esac} \tag{5}$$

When b is *true*, e or e' are non-deterministically assigned to $s(v)$. Otherwise, if A is a mandatory feature and $\boldsymbol{\alpha}^{\text{join}}(\neg A) = \mathit{false}$, we have:

$$s(v) := \texttt{case}\ \ b : e; \mathit{TRUE} : e';\ \ \texttt{esac}$$

In $\boldsymbol{\alpha}^{\text{join}}(M)^{\text{must}}$, if $\widetilde{\boldsymbol{\alpha}^{\text{join}}(A)} = \mathit{false}$ and $\widetilde{\boldsymbol{\alpha}^{\text{join}}(\neg A)} = \mathit{false}$ (that is, A is an optional feature), the above assignment becomes:

$$s(v) := \texttt{case}\ \ \neg b : e'; \mathit{TRUE} : v;\ \ \texttt{esac}$$

Otherwise, if A is a mandatory feature and $\widetilde{\boldsymbol{\alpha}^{\text{join}}(A)} = \mathit{true}$, we have:

$$s(v) := \texttt{case}\ \ b : e; \mathit{TRUE} : e';\ \ \texttt{esac}$$

For example, given the composed model M_1 in Fig. 7 and the optional feature A_1, the models $\boldsymbol{\alpha}^{\text{join}}(M_1)^{\text{may}}$ and $\boldsymbol{\alpha}^{\text{join}}(M_1)^{\text{must}}$ are shown in Figs. 8 and 9, respectively. We can check that $\boldsymbol{\alpha}^{\text{join}}(M_1)^{\text{may}}$ satisfies the properties Φ_1 and Φ_2, whereas $\boldsymbol{\alpha}^{\text{join}}(M_1)^{\text{must}}$ satisfies Φ_2 and Φ_3.

Proposition 1 *Let M be a composed* NuSMV *model obtained using a basic model and features $A_1, \ldots, A_n$. Then, we have $\boldsymbol{\alpha}^{join}([\![M]\!])^{may}_{MTS} = [\![\boldsymbol{\alpha}^{join}(M)^{may}]\!]$, and $\boldsymbol{\alpha}^{join}([\![M]\!])^{must}_{MTS} = [\![\boldsymbol{\alpha}^{join}(M)^{must}]\!]$.*

```
MODULE main
  VAR x : 0..1;
    nA1 : boolean;
  ASSIGN
    init(x) := 0;
    init(nA1) := {TRUE,FALSE};
    next(x) := case nA1 : {x + 1 mod 2, 0};
                     TRUE : 0;
                  easc;
     next(nA1) := case
           nA1 : {next(x)=x+1?FALSE : nA1, nA1};
           TRUE : nA1;
              easc;
   SPEC AF(x ≥ 0);
   SPEC AG(EF(x = 0));
```

Fig. 8 The abstract model $\boldsymbol{\alpha}^{\text{join}}(M_1)^{\text{may}}$

```
MODULE main
  VAR x : 0..1;
     nA1 : boolean;
  ASSIGN
    init(x) := 0;
    init(nA1) := {TRUE,FALSE};
    next(x) := case ¬nA1 : 0; TRUE : x;  esac;
    next(nA1) := nA1;
  SPEC AG(EF(x = 0));
  SPEC EG(x ≥ 0);
```

Fig. 9 The abstract model $\boldsymbol{\alpha}^{\text{join}}(M_1)^{\text{must}}$

Proof It follows from the construction of $[\![M]\!]$, $\boldsymbol{\alpha}^{\text{join}}([\![M]\!])$, $\boldsymbol{\alpha}^{\text{join}}(M)^{\text{may}}$, and $\boldsymbol{\alpha}^{\text{join}}(M)^{\text{must}}$. W.l.o.g., we assume that there is only one feature A.

Consider the case when A is an optional feature and in its CHANGE section, we have IF (b) THEN IMPOSE $s(v) := e$. Then, in $[\![M]\!]$ the assignment is given in Eq. (4), so in $\boldsymbol{\alpha}^{\text{join}}([\![M]\!])^{\text{may}}_{\text{MTS}}$, $s(v)$ will be assigned non-deterministically to e or e' when b holds, and to e' otherwise. This is exactly what happens in $[\![\boldsymbol{\alpha}^{\text{join}}(M)^{\text{may}}]\!]$ as shown in Eq. (5). The other cases are similar to show. $\square$

6 Evaluation

We evaluate our abstraction-based technique for verifying CTL properties of reactive variational systems. It consists of carefully devising projections of the configuration space, then applying the join abstraction on each of them, and verifying the obtained abstract models using the standard version of NUSMV. The evaluation aims to show that we can use state-of-the-art single-system model checkers to efficiently verify different variational systems using our technique. In order to do that, we ask the following research questions:

Table 1 Characteristics of our SPL benchmarks

Benchmark	$\lvert\mathbb{F}\rvert$	$\lvert\mathbb{K}\rvert$	LOC	STATES
Synthetic	20	2^{20}	200	$2^{41.81}$
ELEVATOR	9	2^{9}	300	2^{28}
TELEPHONE	7	2^{7}	700	$2^{48.45}$

RQ1: How efficient is our abstraction-based approach compared to the other approaches for verifying variational systems, such as family-based model checking and brute-force enumeration?

RQ2: Can the abstraction-based approach turn some previously infeasible verification tasks of variational systems into feasible ones?

6.1 Experimental setup

To evaluate our approach, we consider three case studies and a dozen of CTL properties. We use a synthetic example to demonstrate specific characteristics of our approach, as well as the ELEVATOR and TELEPHONE models which are standard benchmarks in the SPLE community [4,11,16,38]. Table 1 summarizes relevant characteristics for each benchmark: the number of features ($\lvert\mathbb{F}\rvert$), the number of valid configurations ($\lvert\mathbb{K}\rvert$), the number of lines of code (LOC), and the total number of reachable states in the compiled variability model. Note that we experiment with different versions of the synthetic example that have $\lvert\mathbb{F}\rvert$ ranging from 2 to 25, but we report in Table 1 the characteristics for the maximal version with $\lvert\mathbb{F}\rvert = 20$ that can be handled by the family-based version of NUSMV.[1]

To establish our objectives, we analyse each case study with several properties. We compare: (1) our abstraction-based approach with the standard version of NUSMV as the verification tool versus (2) the plain family-based model checking approach with the family-based version of NUSMV [11] versus (3) the brute-force enumeration approach which verifies all variants one by one with the standard version of NUSMV. 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. We also report CALLS which is the number of times an approach calls NUSMV while performing a verification task. We say that a task is *infeasible* when it is taking more time than the given timeout threshold, which we set on 3 hours. The time computed for brute-force approach is the average of the all features enabled and the all features disabled configurations multiplied with the number of

---

[1] An extended version of NUSMV [11] implements the family-based algorithm for variability models obtained by composing the basic model and all available features.

```
MODULE features
  VAR fA1, fA2 : boolean;
  ASSIGN
    init(fA1) := {TRUE,FALSE};
    next(fA1) := fA1;
    init(fA2) := {TRUE,FALSE};
    next(fA2) := fA2;
MODULE main
  VAR f : features;
      x : 0..3;
      nA1, nA2 : boolean;
   ASSIGN
     init(x) := 0;
     init(nA1) := {TRUE,FALSE};
     init(nA2) := {TRUE,FALSE};
     next(x) := case
                 f.fA2 & nA2 : x+2 mod 4;
                 TRUE : case f.fA1 & nA1 : x+1 mod 4;
                             TRUE : x;
                        easc;
                 easc;
     next(nA1) := case
             f.fA1 & nA1 : next(x)=x+1?FALSE : nA1;
             TRUE : nA1;
                   easc;
     next(nA2) := case
             f.fA2 & nA2 : next(x)=x+2?FALSE : nA2;
             TRUE : nA1;
                   easc;
```

Fig. 10 The composed model M_2

valid configurations. The BDD model checker NuSMV is run with the parameter `-df -dynamic`, which ensures that the BDD package reorders the variables during verification in case the BDD size grows beyond a certain threshold.

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.

6.2 Synthetic example

As an experiment, we have tested limits of family-based model checking with extended (family-based) 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 basic model in Fig. 5. This was done by adding optional features which increase the basic model's variable x by the number corresponding to the given feature. In effect, the state space of resulting models M_n grows exponentially with the number of features, $n = |\mathbb{F}|$. Thus, for families with high variability, the verification tasks quickly become very prohibitive.

For example, the second feature A_2 introduces a new Boolean variable nA_2, which is non-deterministically initialized. The CHANGE section for A_2 is:

```
IF (nA2) THEN IMPOSE next(x) := x+2
  mod (m+1); next(nA2) := next(x)
=x+2?FALSE : nA2
```

When the basic model in Fig. 5 is composed with two features A_1 and A_2, the resulting model M_2 is shown in Fig. 10. Note that in this case the meta-variable m is instantiated to 3.

In Table 2, we compare the performances of checking the three CTL properties ($\varPhi_1 = \forall\Diamond(x \geq k)$, $\varPhi_2 = \forall\Box(\exists\Diamond(x = k))$, and $\varPhi_3 = \exists\Box(x \geq k)$) using three different approaches: brute force, family based, and abstraction based, on this synthetic model when composed with different number of features. For $|\mathbb{F}| = 20$ (for which $|\mathbb{K}| = 2^{20}$ variants), the family-based NuSMV takes around 117 min to verify the above properties, using the state space of 2^{32} states, whereas for $|\mathbb{F}| > 20$ it has not finished the task within three hours. The analysis time to check the above properties using "brute force" with standard NuSMV ascends to almost 10 days for $|\mathbb{F}| = 20$, and to almost 2 years for $|\mathbb{F}| = 25$. On the other hand, if we apply the variability abstraction $\boldsymbol{\alpha}^{\mathrm{join}}$, we are able to verify the above properties by only one call to standard NuSMV on the *abstract* model in 0.99 s for $\varPhi_1$, 1.07 s for $\varPhi_2$, and 0.12 s for $\varPhi_3$ when $|\mathbb{F}| = 20$, whereas it takes 133 s for $\varPhi_1$, 159 s for $\varPhi_2$, and 0.15 s for $\varPhi_3$ when $|\mathbb{F}| = 25$, thus effectively eliminating the exponential blowup (addresses **RQ1** and **RQ2**). The state space is around 2^{27} for the may-component and 2^{20} for the must-component of the abstract model when $|\mathbb{F}| = 20$, whereas 2^{33} for the may-component and 2^{25} for the must-component of the abstract model when $|\mathbb{F}| = 25$.

6.3 ELEVATOR

The Elevator, designed by Plath and Ryan [38], contains about 300 LOC of NuSMV code and 9 independent optional features that modify the basic behaviour of the elevator. The features are: `Antiprunk`, `Empty`, `Exec`, `OpenIfIdle`, `Overload`, `Park`, `QuickClose`, `Shuttle`, and `TTFull`, thus yielding $2^9 = 512$ variants. The basic Elevator system consists of a single lift that travels between five floors. It has three modules: *`main`*, *`lift`*, and *`button`*. The *`main`* module declares five platform buttons and a single lift, while the *`lift`* module declares variables *floor*, *door*, *direction*, and a further five cabin buttons. The *`button`* module contains a *pressed* variable, which is modelled non-deterministically, and a pressed button remains pressed until the lift has served the floor and its door opened. The lift will always serve all requests in its current direction before it stops and changes direction. When serving a floor, the lift door opens and closes again.

Table 2 Performance results for verifying synthetic models $M_{|\mathbb{F}|}$ using brute-force versus family-based versus abstraction-based approach

$\lvert\mathbb{F}\rvert$	BRUTE FORCE			FAMILY BASED			ABSTRACTION BASED		
	Φ_1	Φ_2	Φ_3	Φ_1	Φ_2	Φ_3	Φ_1	Φ_2	Φ_3
5	2.68	2.76	2.78	0.11	0.12	0.12	0.07	0.13	0.07
10	101.8	128.1	105.4	0.38	0.41	0.37	0.11	0.18	0.09
15	9175	9338	9080	71.8	75.6	71.6	0.25	0.36	0.10
20	Infeasible	Infeasible	Infeasible	7055	7312	7101	0.99	1.07	0.12
25	Infeasible	Infeasible	Infeasible	Infeasible	Infeasible	Infeasible	133	159	0.15

All times are in s (seconds)

Fig. 11 Verification of ELEVATOR properties using tailored abstractions. We compare brute-force versus family-based versus abstraction-based approach. All times are in sec (seconds)

prop- *-erty*	BRUTE-FORCE CALLS	BRUTE-FORCE TIME	FAMILY-BASED CALLS	FAMILY-BASED TIME	ABSTRACTION-BASED CALLS	ABSTRACTION-BASED TIME
Φ_1	512	247.1	1	36.73	2	2.59
Φ_2	512	265.9	1	35.89	2	6.95
Φ_3	512	568.5	1	54.76	1	1.67
Φ_4	512	136.1	1	2.65	2	1.04
Φ_5	512	361.2	1	37.76	2	2.62

First, we consider two properties from ∀CTL. The property "$\Phi_1 = \forall\Box(floor = 2 \wedge liftBut5.pressed \wedge direction = up \Rightarrow \forall[direction = up \, \mathsf{U} \, floor = 5]$" 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 $\boldsymbol{\alpha}^{\text{join}}(\pi_{\llbracket \texttt{Overload} \rrbracket}(\text{ELEVATOR}))^{\text{may}}$ and $\boldsymbol{\alpha}^{\text{join}}(\pi_{\llbracket \neg\texttt{Overload} \rrbracket}(\text{ELEVATOR}))^{\text{may}}$. 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 ∀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 $\boldsymbol{\alpha}^{\text{join}}(\pi_{\llbracket \texttt{Shuttle} \rrbracket}(\text{ELEVATOR}))^{\text{may}}$ and obtain a counter-example for $\boldsymbol{\alpha}^{\text{join}}(\pi_{\llbracket \neg\texttt{Shuttle} \rrbracket}(\text{ELEVATOR}))^{\text{may}}$.

Next, we consider a property from ∃CTL: "$\Phi_3 = (\texttt{OpenIfIdle} \wedge \neg\texttt{QuickClose}) \implies \exists\Diamond(\exists\Box(door = open))$", which states that there exists an execution such that from some state on the door stays open. The property is satisfied for variants where the feature `OpenIfIdle` (when idle, the lift opens its doors) is enabled and `QuickClose` (the lift door cannot be kept open by holding the platform buttons) is disabled. We can verify that Φ_3 holds for $\boldsymbol{\alpha}^{\text{join}}(\pi_{\llbracket \texttt{OpenIfIdle} \wedge \neg\texttt{QuickClose} \rrbracket}(\text{ELEVATOR}))^{\text{must}}$.

The following two properties are neither in ∀CTL nor in ∃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 $\boldsymbol{\alpha}^{\text{join}}(\text{ELEVATOR})^{\text{may}}$ and $\boldsymbol{\alpha}^{\text{join}}(\text{ELEVATOR})^{\text{must}}$ 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 analysing $\boldsymbol{\alpha}^{\text{join}}(\pi_{\llbracket \texttt{Park} \rrbracket}(\text{ELEVATOR}))^{\text{may}}$ and $\boldsymbol{\alpha}^{\text{join}}(\pi_{\llbracket \texttt{Park} \rrbracket}(\text{ELEVATOR}))^{\text{must}}$ using two calls to standard NUSMV.

The size of the family-based version of the ELEVATOR model is 2^{28} states. On the other hand, the sizes of $\boldsymbol{\alpha}^{\text{join}}(\text{ELEVATOR})^{\text{may}}$ and $\boldsymbol{\alpha}^{\text{join}}(\text{ELEVATOR})^{\text{must}}$ are 2^{20} and 2^{19} states, resp. We obtain the similar sizes for individual models used in the brute-force approach. We can see in Fig. 11 that abstractions achieve significant speed-ups between 2.5 and 32 times (resp., 38 and 340 times) faster than the family-based (resp., brute-force) approach (addresses **RQ1**).

6.4 TELEPHONE

The TELEPHONE variational system is initially designed by Plath and Ryan [38] and later extended by Ben-David et. al. [4]. It contains about 700 LOC of NUSMV code and 7 independent optional features, thus yielding $2^7 = 128$ variants. The features are: `Call Forward on Busy` for one phone (`CFB-1`), `Call Forward on Busy` for two phones (`CFB-2`), `Call Forward on No Replay` for one phone (`CFNR-1`), `Call Forward on No Replay` for all phones (`AllCFNR`), `Terminating Call Screening` for one phone (`TCS-1`), `Ring Back When Free` for one phone (`RBWF-1`), as well as `Call Forward Unconditional` for one phone (`CFU-1`).

The basic TELEPHONE system is a network of four synchronous phones, such that two are complete phones: one is terminating phone (it can only receive calls), and one is originating phone (it can only make calls). The module corresponding to each of the phones declares two variables st and $dialled$. Each phone can be found in one of the following states: $st \in \{idle, dialt, trying, busyt, ringingt, talking, ended, talked, ringing\}$. Initially, the phone is in the state $idle$, and from there it may move to $ringing$ (if someone rings to it) or to $dialt$ (if the phone is lifted to dial). The phone is in the state $talking$ if it is in conversation which it initiated, whereas it is in the state $talked$ if the phone is in conversation that was initiated by someone else. $Ended$ means that the phone has hung up the conversation. The variable $dialled$ determines the other phone with which the given phone wants to establish a connection.

A desired property of the telephone system is: a phone can be talked to. That is, if we instantiate this property for the first phone, we have: "$\Phi_1 = \exists\Diamond\,(ph_1.st = talked)$". However, this property is violated by variants with enabled feature `CFU-1` (all calls to the subscriber's phone are diverted to another phone). Therefore, we can tailor an abstraction for verifying this ∃CTL property against two abstract models: $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\texttt{CFU-1}]\!]}(\text{T}\textsc{elephone}))^{\text{must}}$ which violates Φ_1, and $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg\texttt{CFU-1}]\!]}(\text{T}\textsc{elephone}))^{\text{must}}$ which satisfies Φ_1. The next property is "$\Phi_2 = \forall\Box\big(\neg(ph_1.cfu - forw = 0) \implies \forall\Box\neg(ph_1.st \in \{ringing, talked\})\big)$", which states that if a given phone has a forwarding number, then that phone will never ring. This ∀CTL property is violated by variants for which the feature `CFU-1` is disabled. We call the standard NUSMV to check Φ_2 on two abstract models $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\texttt{CFU-1}]\!]}(\text{T}\textsc{elephone}))^{\text{may}}$ and $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg\texttt{CFU-1}]\!]}(\text{T}\textsc{elephone}))^{\text{may}}$. The first abstracted projection satisfies Φ_2, while the second abstracted projection violates Φ_2 and a counter-example is reported.

We consider the property "$\Phi_3 = \forall\Box\big((ph_1.rbwf - number = 2 \wedge ph_1.st = talking \wedge ph_1.dialled = 2) \implies \forall\Diamond(ph_1.rbwf - number = 0)\big)$", in order to confirm the correctness of the feature `RBWF` (if we get a busy tone on calling another phone, there will be an attempt to establish a connection with that phone as soon as it becomes idle). The property Φ_3 states that the stored number will be reset when a call between a phone with `RBWF` feature on and the phone with the stored number is established. This property holds for all variants, and we can successfully verify it using our approach by checking $\boldsymbol{\alpha}^{\text{join}}(\text{T}\textsc{elephone})^{\text{may}}$. The property "$\Phi_4 = \forall\Box\big(ph_1.tcs3 \implies \forall\Box\neg(ph_3.dialled = 1 \wedge ph_3.st \in \{ringingt, talking\})\big)$" states that calls from numbers on the screening list are never accepted (the Boolean variable $tcs3$ is $true$ when the phone 3 is on the screening list of the subscriber's phone). This property is satisfied only by variants with enabled `TCS-1` feature (calls to the subscriber's phone from any number on its screening list will be rejected). We can successfully verify Φ_4 for $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\texttt{TCS-1}]\!]}(\text{T}\textsc{elephone}))^{\text{may}}$ and obtain a counter-example for $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg\texttt{TCS-1}]\!]}(\text{T}\textsc{elephone}))^{\text{may}}$.

The size of the family-based version of the TELEPHONE variability model is around 2^{48} states, whereas the sizes of $\boldsymbol{\alpha}^{\text{join}}(\text{T}\textsc{elephone})^{\text{may}}$, $\boldsymbol{\alpha}^{\text{join}}(\text{T}\textsc{elephone})^{\text{must}}$, as well as individual variants are around 2^{42} states. Figure 12 shows that abstraction-based approach achieves speed-ups between 1.1 and 11 times compared to the family-based approach, and between 15 and 390 times compared to the brute-force approach (addresses **RQ1**).

6.5 Threats to validity

Regarding internal validity, all experiments were executed five times on the same machine with all reported results averaged. The correctness of our abstraction-based approach was shown theoretically (Theorem 1, Theorem 2, and Proposition 1). For the correctness of the implementation, we rely on the results obtained from the family-based model checking approach and brute-force enumeration.

Regarding external validity, results on other case studies may differ and we cannot predict to what extent the obtained results can be generalized in such cases. However, we used benchmarks from published work and observed an improvement in efficiency for various interesting properties.

7 Future extensions

We now give an overview of two possible ways to extend our verification procedure in future. First, we may consider the richer set of temporal properties, as expressed in the modal μ-calculus [33]. Second, we may convert our procedure from manual (where a verification engineer decides what is the most suitable divide-and-conquer strategy for the given verification task) into fully automatic.

7.1 Modal μ-calculus properties

The modal μ-calculus logic [33], denoted L_μ, is a powerful temporal logic which is more expressive than CTL*. L_μ formulae φ are defined by the following grammar:

$$\varphi ::= a \mid \neg a \mid x \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \Box\varphi \mid \Diamond\varphi \mid \mu x.\varphi \mid \nu x.\varphi$$

Fig. 12 Verification of TELEPHONE properties using tailored abstractions. We compare brute-force versus family-based versus abstraction-based approach. All times are in sec (seconds)

prop- *-erty*	BRUTE-FORCE		FAMILY-BASED		ABSTRACTION-BASED	
	CALLS	TIME	CALLS	TIME	CALLS	TIME
Φ_1	128	53760	1	1566	2	138
Φ_2	128	55744	1	3876	2	3526
Φ_3	128	53614	1	1516	1	519
Φ_4	128	51072	1	2451	2	2358

where $x \in Var$ ranges over propositional variables. Note that L_μ formulae φ are given in negation normal form. The formula $\Box\varphi$ expresses that φ is true for every (immediate) successor, whereas $\Diamond\varphi$ expresses that there exists at least one successor for which φ is true. A propositional variable $x \in Var$ is a formula whose meaning (set of states in which it holds) depends on some environment $\rho : Var \to 2^S$ that binds variables to sets of states. The formula $\mu x.\varphi$ (resp., $\nu x.\varphi$) is the least (resp., greatest) fixpoint operator, which represents the smallest (resp., greatest) set x of states in which φ holds (where φ depends on x). The universal and existential fragments $\Box L_\mu$ and $\Diamond L_\mu$ are subsets of L_μ in which the only allowed next-state operators are $\Box$ and $\Diamond$, respectively.

We now formalize the semantics $[\![\varphi]\!]^{\mathcal{T}}_{\rho}$ of a L_μ formula φ over a TS $\mathcal{T}$ and an environment $\rho : Var \to 2^S$, which specifies the interpretation of propositional variables.

Definition 6 The function $[\![\varphi]\!]^{\mathcal{T}}_{\rho} : L_\mu \times (Var \to 2^S) \to 2^S$, which maps a formula φ to the set of states in which it holds, is defined as:

(1) $[\![a]\!]^{\mathcal{T}}_{\rho} = \{s \in S \mid a \in L(s)\}$; $[\![\neg a]\!]^{\mathcal{T}}_{\rho} = \{s \in S \mid a \notin L(s)\}$

(2) $[\![\varphi_1 \wedge \varphi_2]\!]^{\mathcal{T}}_{\rho} = [\![\varphi_1]\!]^{\mathcal{T}}_{\rho} \cap [\![\varphi_2]\!]^{\mathcal{T}}_{\rho}$; $[\![\varphi_1 \vee \varphi_2]\!]^{\mathcal{T}}_{\rho} = [\![\varphi_1]\!]^{\mathcal{T}}_{\rho} \cup [\![\varphi_2]\!]^{\mathcal{T}}_{\rho}$

(3) $[\![\Box\varphi]\!]^{\mathcal{T}}_{\rho} = \{s \in S \mid \forall s' \in S.(s, \lambda, s') \in trans \Rightarrow s' \in [\![\varphi]\!]^{\mathcal{T}}_{\rho}\}$
$[\![\Diamond\varphi]\!]^{\mathcal{T}}_{\rho} = \{s \in S \mid \exists s' \in S.(s, \lambda, s') \in trans \wedge s' \in [\![\varphi]\!]^{\mathcal{T}}_{\rho}\}$

(4) $[\![\mu x.\varphi]\!]^{\mathcal{T}}_{\rho} = \bigcap\{S' \subseteq S \mid [\![\varphi]\!]^{\mathcal{T}}_{\rho[x \mapsto S']} \subseteq S'\}$
$[\![\nu x.\varphi]\!]^{\mathcal{T}}_{\rho} = \bigcap\{S' \subseteq S \mid S' \subseteq [\![\varphi]\!]^{\mathcal{T}}_{\rho[x \mapsto S']}\}$

where $\rho[x \mapsto S']$ is the environment which is the same as ρ, except that x is mapped to S'. For a closed formula φ, we write $\mathcal{T}, s \models \varphi$ for $s \in [\![\varphi]\!]^{\mathcal{T}}_{\bot_{env}}$, where $\bot_{env}$ maps every $x \in Var$ to $\emptyset$. We write $\mathcal{T} \models \varphi$, iff all its initial states satisfy the formula: $\forall s_0 \in I. \mathcal{T}, s_0 \models \Phi$.

We say that an FTS $\mathcal{F}$ *satisfies* a μ-calculus formula φ, written $\mathcal{F} \models \varphi$, iff all its valid variants satisfy the formula: $\forall k \in \mathbb{K}. \pi_k(\mathcal{F}) \models \varphi$.

The semantics of L_μ over an MTS $\mathcal{M}$ is slightly different from the above Definition 6 where a TS $\mathcal{T}$ is considered. In particular, the clause (3) is replaced by:

(3') $[\![\Box\varphi]\!]^{\mathcal{M}}_{\rho} = \{s \in S \mid \forall s'.(s, \lambda, s') \in trans^{\text{may}} \Rightarrow s' \in [\![\varphi]\!]^{\mathcal{M}}_{\rho}\}$
$[\![\Diamond\varphi]\!]^{\mathcal{M}}_{\rho} = \{s \in S \mid \exists s'.(s, \lambda, s') \in trans^{\text{must}} \wedge s' \in [\![\varphi]\!]^{\mathcal{M}}_{\rho}\}$

We then show that the MTS $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F})$ preserves the modal μ-calculus.

Theorem 3 *(Preservation results) For any FTS $\mathcal{F}$:*

($\Box L_\mu$) *For any $\varphi \in \Box L_\mu$, $\boldsymbol{\alpha}^{join}(\mathcal{F})^{may} \models \varphi \implies \mathcal{F} \models \varphi$.*
($\Diamond L_\mu$) *For any $\varphi \in \Diamond L_\mu$, $\boldsymbol{\alpha}^{join}(\mathcal{F})^{must} \models \varphi \implies \mathcal{F} \models \varphi$.*
(L_μ) *For any $\varphi \in L_\mu$, $\boldsymbol{\alpha}^{join}(\mathcal{F}) \models \varphi \implies \mathcal{F} \models \varphi$.*

Proof We prove the most general case (L_μ).

By induction on the structure of φ. All cases except $\Box$ and $\Diamond$ next-operators are straightforward.

For $\varphi = \Box\varphi'$, we proceed by contraposition. Assume $\mathcal{F} \not\models \Box\varphi'$. Then, there exists a configuration $k \in \mathbb{K}$ and a transition $(s_0, \lambda, s_1) \in trans$ of $\pi_k(\mathcal{F})$ (where $s_0 \in I$ of $\pi_k(\mathcal{F})$), such that $s_1 \not\models \varphi'$. By Lemma 2(i), we have that $(s_0, \lambda, s_1) \in trans^{\text{may}}$ of $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F})$, and so $\alpha(\mathcal{F}) \not\models \Box\varphi'$.

For $\varphi = \Diamond\varphi'$. Assume $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F}) \models \Diamond\varphi'$. This means that there exists a must-transition $(s_0, \lambda, s_1) \in trans^{\text{must}}$ of $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F})$ (where $s_0 \in I$ of $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F})$), such that $s_1 \models \varphi'$. By Lemma 2(ii), we have for all $k \in \mathbb{K}$, $(s_0, \lambda, s_1) \in trans$ of $\pi_k(\mathcal{F})$, and so $\pi_k(\mathcal{F}) \models \Diamond\varphi'$. It follows $\mathcal{F} \models \Diamond\varphi'$. $\square$

The preservation result means that abstract models $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F})$ can be used to show validity of modal μ-calculus properties of concrete variational systems.

Similarly as for CTL, in future we can implement a verification procedure for modal μ-calculus. The implementation can be based on the general-purpose mCRL2 model checker, for which it has already been shown how to perform family-based model checking of modal μ-calculus properties [41].

7.2 An automatic verification procedure

We assume that a user of our approach has a good knowledge of the given variability model and property, so that he can manually devise suitable projections (partitionings) of the configuration space and variability abstractions before verification. We now give an overview of an algorithm, which aims to automate our verification approach so that the appropriate partitionings of the configuration space are constructed automatically. The algorithm is based on an abstraction and refinement framework for CTL⋆ properties, which iteratively refines abstract variability models until either a genuine counter-example is found or the property satisfaction is shown for all variants. In the heart of this algorithm are 3-valued model checking games [39,40], which represent the most suitable framework for defining the refinement. In particular, we use Shoham–Grumberg algorithm for solving such games, which is able to verify a CTL* property Φ on an MTS $\mathcal{M}$, that is, to check $\mathcal{M} \models \Phi$?. Since the 3-valued semantics of CTL* over MTSs is considered, there are three possible outcomes of the above check: (1) $\mathcal{M}$ satisfies Φ; (2) $\mathcal{M}$ does not satisfy Φ; and (3) an indefinite (don't know) result. In the case of an indefinite (don't know) result, the games framework [39,40] provides a way to find the failure reason for it which can be used for defining a refinement criterion. It splits abstract configurations so that the new, refined abstract configurations represent smaller subsets of concrete configurations. The sketch of the automatic abstraction-refinement procedure for checking $\mathcal{F} \models \Phi$, where $\mathbb{K}$ is the set of configurations, is as follows:

1. Check by the algorithm for solving 3-valued model checking games whether $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F}) \models \Phi$?
2. If the result is *true*, Φ is satisfied by all variants in $\mathbb{K}$.
3. If the result is *false*, Φ is violated by all variants in $\mathbb{K}$.
4. Otherwise, an indefinite result is returned. Let the may-transition $s_1 \xrightarrow{\lambda} s_2$ in $\boldsymbol{\alpha}^{\text{join}}(\mathcal{F})$ be the reason for failure (as identified by the game-based model checking algorithm), and let ψ be the feature expression guarding this transition in $\mathcal{F}$. We split the configuration set $\mathbb{K}$ into two subsets $\mathbb{K} \cap [\![\psi]\!]$ and $\mathbb{K} \cap [\![\neg\psi]\!]$. We go back to Step (1), to check $\pi_{[\![\psi]\!]}(\mathcal{F}) \models \Phi$ with the set of configurations $\mathbb{K} \cap [\![\psi]\!]$, and to check $\pi_{[\![\neg\psi]\!]}(\mathcal{F}) \models \Phi$ with the set of configurations $\mathbb{K} \cap [\![\neg\psi]\!]$.

In future, we can develop more precisely and implement such an automatic abstraction-refinement procedure for verifying CTL* properties of variational systems. In this way, we will establish a brand new connection between games and SPL communities.

8 Related work

Many family-based analysis and verification techniques that work on the level of variational systems and programs have been proposed in recent years (see [43] for survey). Some successful examples range from family-based syntax and type checking [27,31,32], to family-based static analysis [5,19–22,37] and family-based verification by simulation [29,30,44]. Family-based model checking has also been an active research field, where different approaches have been developed for verifying variational systems.

One of the earliest attempts for modelling variability and variational systems is by using modal transition systems (MTSs) [34,42], where optional 'may'-transitions are used to model variability. In contrast, here we use MTSs with an entirely different goal of abstracting variational systems, which is closer to the original idea of introducing MTSs by Larsen and Thomsen [35] (abstraction in system modelling and verification). Beek et al. [42] have implemented a model checking tool, called VMC, for verifying variability models expressed as MTSs and properties expressed as v-ACTL formulae. They use MTSs as a compact representation of a family of Labelled TSs, and use variability-aware action-based CTL logic to reason over MTSs. The preservation result in [42] shows that if a property holds over an MTS, then the same property holds for all variants derived from that MTS. In contrast, here we use MTSs to represent abstract models of a family of systems, and our preservation result (Theorem 1) states that if a property holds over an abstract model, then it holds over all variants derived from the concrete family. Subsequently, various variability models have been developed. Ultimately, the popular feature transition systems (FTSs) have been introduced by Classen et al. [10], which are today widely accepted as models essentially sufficient for most purposes of family-based model checking of variational systems.

Firstly, Classen et al. [9] have presented specifically designed (explicit) family-based model checking algorithms for verifying FTSs against LTL properties, which are implemented in the $\overline{\text{SNIP}}$ model checker. Then, (symbolic) family-based model checking algorithms [8,11] have been proposed to enable verification of FTSs against CTL properties, which are implemented as an extension of the BDD model checker NuSMV. It uses symbolic encoding for FTSs as well as symbolic algorithms for their verification. One of the most prominent methods to make all these approaches based on FTSs more scalable to larger systems is to apply abstractions. In this work, we show how to construct abstract models of FTSs that preserve CTL* properties. For implementation, we use the standard version of the BDD model checker NuSMV, where abstract models are symbolically encoded and its symbolic algorithms are used for model checking.

Simulation-based abstractions on FTSs introduced in [13,14] are defined by using existential F-abstraction functions, and simulation relation is used to relate different abstraction levels. Different levels of precision of so-called feature abstractions in [14] are defined by simply enriching (resp., reducing) the sets of variants for which transitions are enabled. These abstractions are applied either on concrete FTSs [13], or on an intermediate concrete semantic model (called featured program graph) [14]. Therefore, they report smaller efficiency gains. For example, the approach [13] results in marginal efficiency reductions of verification times of 8–9 % compared to the unabstracted approach. On the other hand, our variability abstractions defined as Galois connections are capable to change not only the feature expression labels of transitions but also the sets of available features and valid configurations. Moreover, we can also use projection, which partitions the configuration space, in order to build various more sophisticated verification strategies. We apply our abstractions as preprocessor transformations directly on high-level modelling languages, thus avoiding to generate and store any concrete model in the memory. Therefore, we report significant performance speed-ups compared to the unabstracted approach.

All the previous abstractions applied on FTSs [13,14, 17,18,24] are conservative, and thus, they construct over-approximated abstract models that preserve satisfiability only of LTL and universal ∀CTL properties. To our knowledge, in this work, for the first time, we propose abstractions on FTSs that preserve all CTL⋆ (and μ-calculus), thus significantly extending the previous works on abstractions of FTSs.

The abstraction and refinement procedure for automatic verification of LTL properties of variational systems has been developed in [25]. If a spurious counter-example (introduced due to the abstraction) is found in the abstract model, the procedure [25] uses Craig interpolation to extract relevant information from it in order to define the refinement of abstract models. As we noted in Sect. 7, in the context of CTL⋆ properties, the 3-valued model checking games proposed by Shoham and Grumberg [39,40] represent the most suitable framework to define the refinement [23].

Verifying variability models against properties specified in modal μ-calculus has been also an interesting topic of research. Some examples are model checking algorithms for variability models specified in PL-CCS and Delta-CCS. PL-CCS [28] is an extension of Milner's process algebra CCS, which is enriched with a variant operator as a means to implement variability. Delta-CCS [36] is another delta-oriented extension of CCS, in which variability is achieved by decomposing the product line into a core process and a set of delta modules that encapsulate change directives based on term rewriting semantics. Beek et. al. [41] have also presented an approach for family-based model checking of modal μ-calculus properties using the general-purpose mCRL2 model checker. In this work, we show that the resulting abstract models also preserve the full μ-calculus properties, thus enabling our approach to be used for their verification as well.

Another approach to efficiently verify variational systems is by using variability encoding [2,30], which transforms features into non-deterministically initialized variables (replaces compile time with runtime variability). Then, the generated *family simulator* is verified using the standard single-system model checkers. However, in case of violation, the (single-system) model checker stops after a single counter-example and a violating variant are found. Therefore, this answer is incomplete (limited) since there might be other satisfying variants and also there might be other violating variants with different counter-examples. In contrast, family-based model checking and our approach provide precise conclusive results for all variants in the family.

9 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 projections and abstractions used in our divide-and-conquer verification procedure are implemented as source-to-source transformations of high-level fNUSMV variability models. The evaluation confirms that various CTL properties can be efficiently verified in this way.

In this work, we focus on the state-based approach for analysing variational systems. This means that we abstract from actions, and only use atomic propositions of the states to formulate system properties. A combined action- and state-based approach is possible, but leads to more involved definitions and concepts. Moreover, NUSMV model checker used in the implementation is also based exclusively on the state-based approach.

References

1. Apel, S., Batory, D.S., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines—Concepts and Implementation. Springer, Berlin (2013)
2. Apel, S., Rhein, A., von Wendler, P., Größlinger, A., Beyer, D.: Strategies for product-line verification: case studies and experiments. In: 35th International Conference on Software Engineering, ICSE '13, pp. 482–491. IEEE Computer Society (2013)
3. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press, Cambridge (2008)
4. Ben-David, S., Sterin, B., Atlee, J.M., Beidu, S.: Symbolic model checking of product-line requirements using sat-based methods. In: 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Volume 1, pp. 189–199. IEEE Computer Society (2015)
5. 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)
6. 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, volume 2404 of LNCS, pp. 359–364. Springer (2002)
7. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Logics of Programs, Workshop, 1981, volume 131 of Lecture Notes in Computer Science, pp. 52–71. Springer (1981)
8. 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)
9. Classen, A., Cordy, M., Heymans, P., Legay, A., Schobbens, P.-Y.: Model checking software product lines with SNIP. STTT **14**(5), 589–612 (2012)

10. Classen, A., Cordy, M., Schobbens, P.-Y., Heymans, P., Legay, A., Raskin, J.-F.: Featured transition systems: foundations for verifying variability-intensive systems and their application to LTL model checking. IEEE Trans. Softw. Eng. **39**(8), 1069–1089 (2013)
11. 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)
12. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Boston (2001)
13. Cordy, M., Classen, A., Perrouin, G., Schobbens, P.-Y., Heymans, P., Legay, A.: Simulation-based abstractions for software product-line model checking. In: Glinz, M., Murphy, G.C., Pezzè, M. (eds.) 34th International Conference on Software Engineering, ICSE 2012, pp. 672–682. IEEE (2012)
14. Cordy, M., Heymans, P., Legay, A., Schobbens, P.-Y., Dawagne, B., Leucker, M.: Counterexample guided abstraction refinement of product-line behavioural models. In: Cheung, S.-C. Orso, A., Storey, M.-A.D. (eds.) Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), pp. 190–201. ACM (2014)
15. Cousot, P.: Partial completeness of abstract fixpoint checking. In: Abstraction, Reformulation, and Approximation, 4th International Symposium, SARA 2000, Proceedings, volume 1864 of LNCS, pp. 1–25. Springer (2000)
16. Dimovski, A.S.: Abstract family-based model checking using modal featured transition systems: Preservation of ctl*. In: Fundamental Approaches to Software Engineering—21st International Conference, FASE 2018, Proceedings, volume 10802 of LNCS, pp. 301–318. Springer (2018)
17. 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, volume 9232 of LNCS, pp. 282–299. Springer (2015)
18. 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)
19. 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, volume 37 of LIPIcs, pp. 247–270. Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik (2015)
20. Dimovski, A.S., Brabrand, C., Wasowski, A.: Finding suitable variability abstractions for family-based analysis. In: FM 2016: Formal Methods—21st International Symposium, Proceedings, volume 9995 of LNCS, pp. 217–234 (2016)
21. Dimovski, A.S., Brabrand, C., Wasowski, A.: Variability abstractions for lifted analyses. Sci. Comput. Program. **159**, 1–27 (2018)
22. Dimovski, A.S., Brabrand, C., Wasowski, A.: Finding suitable variability abstractions for lifted analysis. Formal Asp. Comput. **31**(2), 231–259 (2019)
23. Dimovski, A.S., Legay, A., Wasowski, A.: Variability abstraction and refinement for game-based lifted model checking of full CTL. In: Fundamental Approaches to Software Engineering—22nd International Conference, FASE 2019, Proceedings, volume 11424 of LNCS, pp. 192–209. Springer (2019)
24. 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, volume 10460 of LNCS, pp. 249–268. Springer (2017)
25. 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, volume 10202 of LNCS, pp. 406–423 (2017)
26. Ebert, C., Jones, C.: Embedded software: facts, figures, and future. IEEE Comput. **42**(4), 42–52 (2009)
27. 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)
28. Gruler, A., Leucker, M., Scheidemann, K.D.: Modeling and model checking software product lines. In: Formal Methods for Open Object-Based Distributed Systems, 10th IFIP WG 6.1 International Conference, FMOODS 2008, Proceedings, volume 5051 of LNCS, pp. 113–131. Springer (2008)
29. 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 International Conference on Automated Software Engineering, ASE 2015, pp. 597–607 (2015)
30. Iosif-Lazar, A.F., Melo, J., Dimovski, A.S., Brabrand, C., Wasowski, A.: Effective analysis of c programs by rewriting variability. Program. J. **1**(1), 1 (2017)
31. Kästner, C., Apel, S., Thüm, T., Saake, G.: Type checking annotation-based product lines. ACM Trans. Softw. Eng. Methodol. **21**(3), 14 (2012)
32. 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)
33. Kozen, D.: Results on the propositional mu-calculus. Theor. Comput. Sci. **27**, 333–354 (1983)
34. 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, volume 4421 of LNCS, pp. 64–79. Springer (2007)
35. 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)
36. Lochau, M., Mennicke, S., Baller, H., Ribbeck, L.: Incremental model checking of delta-oriented software product lines. J. Log. Algebraic Methods Program. **85**(1), 245–267 (2016)
37. Midtgaard, J., Dimovski, A.S., Brabrand, C., Wasowski, A.: Systematic derivation of correct variability-aware program analyses. Sci. Comput. Program. **105**, 145–170 (2015)
38. Plath, M., Ryan, M.: Feature integration using a feature construct. Sci. Comput. Program. **41**(1), 53–84 (2001)
39. Shoham, S., Grumberg, O.: A game-based framework for CTL counterexamples and 3-valued abstraction-refinement. ACM Trans. Comput. Log. **9**(1), 1 (2007)
40. Shoham, S., Grumberg, O.: Compositional verification and 3-valued abstractions join forces. Inf. Comput. **208**(2), 178–202 (2010)
41. 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, volume 10202 of LNCS, pp. 387–405 (2017)
42. 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. Algebraic Methods Program. **85**(2), 287–315 (2016)
43. 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)

44. von Rhein, A., Thüm, T., Schaefer, I., Liebig, J., Apel, S.: Variability encoding: from compile-time to load-time variability. J. Log. Algebraic Methods Program. **85**(1), 125–145 (2016)