Empirical Software Engineering Journal manuscript No.
(will be inserted by the editor)

Efficient Static Analysis and Verification of Featured Transition Systems

Maurice H. ter Beek · Ferruccio Damiani · Michael Lienhardt · Franco Mazzanti · Luca Paolini

Received: date / Accepted: date

Abstract A Featured Transition System (FTS) models the behaviour of all products of a Software Product Line (SPL) in a single compact structure, by associating action-labelled transitions with features that condition their presence in product behaviour. It may however be the case that the resulting featured transitions of an FTS cannot be executed in any product (so called *dead transitions*) or, on the contrary, can be executed in all products (so called *false optional transitions*). Moreover, an FTS may contain states from which a transition can be executed only in some products (so called *hidden deadlock states*). It is useful to detect such ambiguities and signal them to the modeller, because dead transitions indicate an anomaly in the FTS that must be corrected, false optional transitions indicate a redundancy that may be removed, and hidden deadlocks should be made explicit in the FTS to improve the understanding of the model and to enable efficient verification—if the deadlocks in the products should not be remedied in the first place. We provide an algorithm to analyse an FTS for ambiguities and a means to transform an ambiguous FTS into an unambiguous one. The scope is twofold: an ambiguous model is typically undesired as it gives an unclear idea of the SPL and, moreover, an unambiguous FTS can efficiently be model checked. We empirically show the suitability of the algorithm by applying it to a number of benchmark SPL examples from the literature, and we show how this facilitates a kind of family-based model checking of a wide range of properties on FTSs.

M.H. ter Beek✉, F. Mazzanti
ISTI–CNR, Via Giuseppe Moruzzi 1, Pisa, 56124 Italy
E-mail: maurice.terbeek@isti.cnr.it, franco.mazzanti@isti.cnr.it

F. Damiani, L. Paolini
University of Turin, Corso Svizzera 185, 10149 Turin, Italy
E-mail: ferruccio.damiani@unito.it, luca.paolini@unito.it

M. Lienhardt
ONERA, Chemin de la Vauve aux Granges 6, 91123 Palaiseau, France
E-mail: michael.lienhardt@onera.fr

Keywords Software product lines · Formal specification · Behavioural model · Featured transition systems · Static analysis · Formal verification

1 Introduction

Software Product Line Engineering (SPLE) advocates the reuse of components (systems as well as software) throughout all phases of product development. Following this paradigm, businesses today no longer develop single products, but families or product lines of closely-related, customisable products. Upon identifying the relevant features of the product domain, to exploit their commonality and variability, a feature diagram or feature model defines those combinations of features that constitute valid product configurations [2]. The automated analysis of such variability models has a 30-year history [26,89]. Think, e.g., of the detection of anomalies like so called dead or false optional features. Behavioural models with variability, on the other hand, have a shorter history [67,65,76,66,68,3,78] and they have received considerable attention only during the last decade, following the seminal paper by Classen et al. [40]. SPLs often concern massively (re)used critical software (e.g., in smartphones and the automotive industry), thus it is important to demonstrate their correct behaviour next to their correct configuration.

A Featured Transition System (FTS) is a formal model with variability for capturing the behaviour of all products of an SPL in one compact model [39, 44]; its action-labelled transitions are associated with features that condition their presence in product behaviour. Proving correctness of such models through model checking or testing is challenging. Ideally, the compact structure of the FTS is exploited to reason on the whole SPL at once. Such an *all-in-one* technique, according to which the behaviour of all products is examined only once simultaneously, is called family-based analysis in contrast to a brute force enumerative product-based analysis, according to which the behaviour of every product is examined individually, *one-by-one* [88]. Over the past decade, FTSs have shown to be amenable to family-based testing and model-checking [74,37,42,39,38,51,12,60,54,24,62,57,18].

In [9], we tackled the automated static analysis of FTSs. We defined the following three ambiguities for an FTS: a *dead transition* (i.e., a featured transition that is unreachable, and thus cannot be executed, in any product); a *false optional transition* (i.e., a featured transition that can be executed in all products in which its source state is reachable); and a *hidden deadlock state* (i.e., a state from which a transition can be executed only in some products). We developed an algorithm to detect ambiguities in FTSs (and a means to resolve them), mimicking the well-established anomaly detection for feature models, with a proof of its correctness. The motivations we presented in [9] were twofold: an ambiguous FTS is often undesired, since it gives an unclear idea of the SPL behaviour, and an unambiguous FTS paves the way for an efficient kind of family-based model checking. We illustrated the latter on a few examples from the literature.

This paper extends [9] in the following ways.

1. We introduce an engineering methodology aimed towards providing feedback to SPL modellers to possibly improve their FTS models and, subsequently, a strategy which offers a number of verification options (cf. Fig. 4). A dead transition in an FTS indicates a modelling error that must be corrected. A false optional transition indicates a redundancy that may be intentional, but resolving it allows for more efficient verification options. A hidden deadlock should be made explicit in the model to improve understanding and to enable an efficient kind of family-based verification—if the deadlocks in the products that are the cause should not be remedied in the first place.
2. Driven by the need to improve the practical applicability of our automated static analysis for behavioural ambiguity detection in FTSs, we present a new algorithm (more efficient than that presented in [9]) for detecting ambiguities in FTSs by reducing the analysis to SAT solving. In addition, we prove its correctness.
3. To demonstrate the improved practical applicability, we apply our algorithm to a larger set of benchmark SPL examples than in [9], including the FTS of the complete mine pump model of [35,36] and that of the Claroline SPL of [50] with over 10,000 transitions, both of which are not tractable with the algorithm presented in [9]. We empirically show the suitability of the new algorithm by means of a clear runtime speedup.
4. We capitalise on the promise of an efficient kind of family-based model checking by demonstrating how properties specified in either the well-known Linear-time Temporal Logic (LTL) or in v-ACTLive$^{\Box}$, a rich action-based and variability-aware fragment of the well-known branching-time Computation Tree Logic (CTL), can be verified (with a linear complexity) directly on an unambiguous FTS (ignoring its feature expressions) such that validity is preserved in all LTSs modelling product behaviour. The preservation of valid v-ACTLive$^{\Box}$ properties was anticipated in [9], while the preservation of valid LTL properties was not observed before. These results imply the addition of two efficient verification options to the above mentioned strategy provided to SPL modellers (cf. Fig. 4).

Outline After mentioning some related work in Section 2 and providing some background in Section 3, we provide our engineering methodology in Section 4 by defining ambiguities in FTSs and providing a means to resolve them. In Section 5, we present the new static analysis algorithm to detect ambiguities in FTSs, based on SAT solving, and prove its correctness. In Section 6, we empirically show the suitability of the new algorithm by applying it to a number of exemplary FTSs from the literature. In Section 7, we show the feasibility of an efficient kind of family-based model checking of FTSs made possible by the static analysis algorithm. Finally, we conclude the paper in Section 8.

2 Related Work

Static analysis of FTSs mimics the automated analysis of feature models by defining behavioural counterparts of dead and false optional features [26,89]. It is related to static (program) analysis [86,33], which includes the detection of bugs in the code (like using a variable before its initialisation) but also the identification of code that is redundant or unreachable.

In [74], conventional static analysis techniques are applied to SPLs that are represented in the form of object-oriented programs with feature modules. The aim is to find irrelevant features for a specific test in order to use this information to reduce the effort in testing an SPL by limiting the number of SPL programs to examine to those with relevant features. In [30], several well-known static analysis techniques are lifted to Java-based SPLs without the exponential blowup caused by generating and analysing all products individually. This is achieved by converting such analyses to feature-sensitive analyses that operate on the entire SPL code in one single pass. Basically, if the original analysis reports that a data-flow property holds at a given program statement, then the lifted analysis reports a feature constraint (a logical expression over the set of features) under which that property holds at the given statement.

In [73], static type checking is extended from single programs to an entire SPL (program family) by extending the type system of a subset of Java with feature annotations. This guarantees that whenever the SPL is well-typed, then all possible program variants are well-typed as well, without the need to generate and compile them first. In [48], type-checking for product lines is mechanised and soundness of a constraint-based type system for Lightweight Feature Java (LFJ), an extension of Lightweight Java with support for features, is proved using a full formalisation of LFJ in the Coq proof assistant [28].

An encompassing overview of analysis strategies for SPLs, including type checking, static analysis, model checking, and theorem proving, can be found in [88] and a recent empirical study on applying variability-aware static analysis techniques to real-world configurable systems is presented in [87].

Family-based model checking of behavioural SPL models provides a means to simultaneously verify multiple behavioural product models in a single run. Properties can be verified with dedicated SPL model-checking tools such as SNIP [37,39], ProVeLines [42], VMC [20,19,13], fNuSMV [38,58], ProFeat [34] (for probabilistic model checking), or QFLan [16,90] (for statistical model checking), or—through suitable abstractions or encodings—with well-known classical model checkers like SPIN [61,60,62], PRISM [64] (for probabilistic model checking), Maude [82], mCRL2 [24,18], or NuSMV [57].

In this paper, we introduce an engineering methodology that enables a kind of family-based model checking for FTSs, according to a strategy that is sketched in Figure 4 (the part that is not in red). This figure will be discussed in more detail in Sections 4 and 7. The strategy that is sketched is as follows. If (i) the FTS is live, which is the case whenever it has no hidden deadlocks (so, unambiguous FTSs are live), and (ii) the property ϕ to be verified is specified in either LTL or v-ACTLive$^{\Box}$, then ϕ can be verified directly on the FTS (by ignoring its feature expressions) and if (iii) ϕ holds, this validity is preserved in all LTSs modelling product behaviour, i.e. ϕ holds for all products. If any of these three conditions does not hold, the property needs to be verified with classical (family-based) approaches, such as the ones mentioned above.

The verification methodology depicted in Figure 4 thus indicates specific cases in which verification of live FTSs reduces to verification of corresponding MTSs and LTSs (which, as we will see, can be obtained straightforwardly by ignoring the feature expressions, and distinguishing necessary and optional transitions in case of MTSs) with a linear complexity. However, if either (i) the property to be verified is not a v-ACTLive$^\Box$ or LTL formula, or (ii) the result of the verification is false, then the formula needs to be verified with classical family-based model checking or by means of product-based model checking, with an exponential complexity [39,38].

3 Background

In this section, we provide some background needed for the sequel. Labelled Transition Systems (LTSs) are the underlying behavioural structure of FTSs.

Definition 1 (LTS) A *Labelled Transition System* (LTS) is a quadruple $\mathcal{L} = (S, \Sigma, s_0, \delta)$, where S is a finite (non-empty) set of states, Σ is a set of actions, $s_0 \in S$ is an initial state, and $\delta \subseteq S \times \Sigma \times S$ is a transition relation.

We call $(s, a, s') \in \delta$ an a-(labelled) transition (from source state s to target state s') and we may also write it as $s \xrightarrow{a} s'$.

We recall classical notions for LTSs that will be used throughout the paper.

Definition 2 (reachability) Let $\mathcal{L} = (S, \Sigma, s_0, \delta)$ be an LTS. A sequence $p = s_0 t_1 s_1 t_2 s_2 \cdots$ is a *path* of $\mathcal{L}$ if $t_i = (s_{i-1}, a_i, s_i) \in \delta$ for all $i > 0$; p is said to *visit* states $s_0, s_1, \ldots$ and transitions $t_1, t_2, \ldots$ and we denote its ith state by $p(i)$ and its ith transition by $p\{i\}$.

A state $s \in S$ is *reachable* (via p) in $\mathcal{L}$ if there exists a path p that visits it, i.e., $p(i) = s$ for some $i \geq 0$; s is a *deadlock* if it has no outgoing transitions, i.e., $\nexists\, (s, a, s') \in \delta$, for all $a \in \Sigma$ and $s' \in S$.

A transition $t = (s, a, s') \in \delta$ is *reachable* (via p) in $\mathcal{L}$ if there exists a path p that visits it, i.e., $p\{i\} = t$, for some $i > 0$.

Example 1 In Figure 1, we depict the LTSs $\mathcal{L}_1$ and $\mathcal{L}_2$, modelling the behaviour of two different coffee machines, adapted from [24,9]. Each LTS has actions to insert coins (*ins*) and to pour either standard (*std*) or extra large (*xxl*) coffee upon the insertion of one or two coins, respectively. Clearly all states are reachable and there are no deadlocks.

FTSs were introduced in [40] to concisely model the behaviour of all the products of an SPL, modelled as LTSs, in one transition system by annotating transitions with conditions expressing their presence in (product) LTSs. Let $\mathbb{B} = \{\top, \bot\}$ denote the Boolean constants true ($\top$) and false ($\bot$), and let $\mathbb{B}(F)$

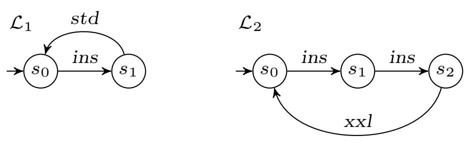


Fig. 1 LTSs $\mathcal{L}_1$ and $\mathcal{L}_2$ modelling coffee machines

denote the set of propositional formulas over a set of features F (i.e., using features as propositional variables). We do not formalise a language for propositional formulas in order to allow the inclusion of all possible propositional connectives but, in particular, we include the constants from $\mathbb{B}$. The elements of $\mathbb{B}(F)$ are also called *feature expressions*. An FTS is an LTS equipped with a feature model and a function that labels each transition with a feature expression. In the following definition, the feature model is represented by the set of its (product) configurations, where each configuration is represented by a Boolean assignment to the features (i.e., selected $= \top$ and unselected $= \bot$).

Definition 3 (FTS) A *Featured Transition System* (FTS) is a sextuple $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$, where S is a finite (non-empty) set of states, Σ is a set of actions, $s_0 \in S$ is the initial state, $\delta \subseteq S \times \Sigma \times \mathbb{B}(F) \times S$ is a transition relation, F is a set of features, and $\Lambda \subseteq \{\, \lambda : F \to \mathbb{B} \,\}$ is a set of *(product) configurations*.

Given a feature expression $\phi \in \mathbb{B}(F)$, we call $(s, a, \phi, s') \in \delta$ *featured transition* (labelled with a and limited to configurations satisfying ϕ) and $(s, a, \top, s') \in \delta$ *must transition*. We may write featured transitions as $s \xrightarrow{a \,|\, \phi} s'$.

The notions from Definition 2 (path, reachability, deadlock) are carried over to FTSs by ignoring the feature expressions.

A configuration $\lambda \in \Lambda$ satisfies a feature expression $\phi \in \mathbb{B}(F)$, denoted by $\lambda \models \phi$, whenever ϕ is valid in the interpretation λ, i.e., the result of substituting the value of the features occurring as variables in ϕ according to λ is $\top$. Thus, by definition, $\lambda \models \top$.

Without loss of generality, in the sequel we only consider FTSs that do not contain two featured transitions $q \xrightarrow{a \,|\, \phi} q'$ and $q \xrightarrow{a \,|\, \phi'} q'$ such that $\phi \neq \phi'$. Any FTS that does not satisfy this criterion can be transformed into one that does by replacing the two transitions with one featured transition $q \xrightarrow{a \,|\, \phi \vee \phi'} q'$.

Definition 4 (product) Let $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$ be an FTS. The LTS specified by a particular configuration $\lambda \in \Lambda$, denoted by $\mathcal{F}|_\lambda$, is called a *product* of $\mathcal{F}$. It is obtained from $\mathcal{F}$ by first removing all featured transitions whose feature expressions are not satisfied by λ (resulting in the LTS $(S, \Sigma, s_0, \delta')$, with $\delta' = \{\, (s, a, s') \mid (s, a, \phi, s') \in \delta \text{ and } \lambda \models \phi \}$), and then removing all unreachable states and their outgoing transitions. Given a featured transition $(s, a, \phi, s') \in \delta$, we call $(s, a, s') \in \delta'$ its *corresponding (LTS) transition*. The set of products of $\mathcal{F}$ is denoted by $\mathsf{lts}(\mathcal{F})$.

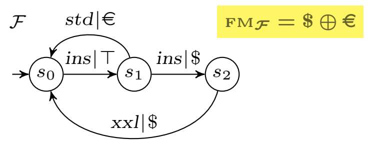


Fig. 2 FTS $\mathcal{F}$ modelling a product line of coffee machines

Note that, by construction: (i) each product does not contain unreachable states or transitions, (ii) each must transition of the FTS has a corresponding transition in the products in which it is reachable, (iii) each product does not contain states or actions that were not originally present in the FTS, and (iv) each featured transition has a unique corresponding LTS transition when its source state is reachable.

The *feature model expression* of $\mathcal{F}$, denoted by $\text{FM}_{\mathcal{F}}$, is a feature expression that represents Λ (like, e.g., the formula in conjunctive normal form $\bigvee_{\lambda \in \Lambda} (\bigwedge_{f \in F} (\{ f \mid \lambda(f) = \top \} \cup \{ \neg f \mid \lambda(f) = \bot \}))$. Thus, for all $\lambda : F \to \mathbb{B}$ it holds that $\lambda \models \text{FM}_{\mathcal{F}}$ if and only if $\lambda \in \Lambda$. We may write FM instead of $\text{FM}_{\mathcal{F}}$ if no confusion can arise.

Example 2 In Figure 2, we depict an FTS $\mathcal{F}$ modelling the behaviour of the two coffee machines from Example 1 as a product line of coffee machines, adapted from [24, 9]. Imagine that extra large coffee is exclusively available for the American market, while standard coffee is exclusively available for the European market. To this aim, $\mathcal{F}$ has transitions labelled with features \$ and €, representing products for either the American or the European market, respectively, and a must transition that must be present in every product. Its feature model, depicted in Figure 3(left), can be represented by the feature expression $\text{FM}_{\mathcal{F}} = \$ \oplus €$, where $\oplus$ denotes the *exclusive disjunction* operation. Hence the product configurations of $\mathcal{F}$ are $\Lambda = \{\lambda_1, \lambda_2\}$, where $\lambda_1(\$) = \bot$, $\lambda_1(€) = \top$, $\lambda_2(\$) = \top$, and $\lambda_2(€) = \bot$. The LTSs $\mathcal{F}|_{\lambda_1} = \mathcal{L}_1$ and $\mathcal{F}|_{\lambda_2} = \mathcal{L}_2$, depicted in Figure 1, model the behaviour of the only two products of $\mathcal{F}$: configuration λ_1 for the European market and λ_2 for the American market.

Parallel composition of FTSs is equal to the classical parallel composition of LTSs modulo projection [36, 39]. Intuitively, parallel composition partially interleaves the transitions of the LTSs, permitting asynchronous execution of their actions, except for those with shared actions, which are synchronised, thus only permitting execution of their actions at the same time. In case of FTSs, the feature expressions of synchronised transitions are conjuncted,[1] while each interleaved transition simply maintains its feature expression [36, 39].

[1] We foresee an optimisation of conjuncted feature expressions to foster useful output (e.g., the synchronisation of two must transitions could lead to a conjuncted feature expression $\top \wedge \top$, which would technically not be a must transition according to Definition 3 and could thus be detected as a false optional transition, as we will see in Definition 6(ii).

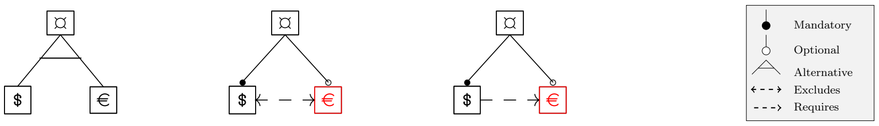


Fig. 3 Feature models of product line of coffee machines (left), with a dead feature (middle), and with a false optional feature (right)

4 Ambiguities in FTSs

When applying automated analysis of feature models, the better known analysis operations that are typically being performed concern the detection of anomalies (cf., e.g., [26,89]). These anomalies reflect ambiguous or even contradictory information. Examples include so-called dead and false optional features. A feature is *dead* if it is not contained in any product configuration of the FTS, whereas it is *false optional* if it is contained in all product configurations of the FTS even though it is not a designated mandatory feature. Such anomalies are typically due to an incorrect use of cross-tree constraints. Consider the feature models depicted in Figure 3. The one on the left corresponds to the feature model expression $\$ \oplus €$ from Example 2 and it has neither dead nor false optional features. The one in the middle corresponds to the feature model expression $\$ \wedge (\$ \uparrow €)$, where $\uparrow$ is the negation of conjunction (a.k.a. *not and*), and it has a dead feature €, indicated in red, because this optional feature is excluded by the mandatory feature $ and thus never present. The feature model on the right, finally, corresponds to the feature model expression $\$ \wedge (\$ \rightarrow €)$, meaning that € is false optional, indicated in red, because it is required by the mandatory feature $ and as such always present.

In this section, we formalise equivalent notions in a behavioural setting, by adapting the above notions to (featured) transitions of an FTS (Section 4.1). Furthermore, we define ambiguous FTSs and we show how to transform any ambiguous FTS into an unambiguous one (Section 4.2). This constitutes our envisioned engineering methodology, which is sketched in Figure 4 (the top-right red part) together with a number of verification options (the part of Fig. 4 that is not red) organised in a strategy that was briefly outlined in Section 2 and which will be discussed in more detail in Section 7. This engineering methodology improves the clarity of behavioural SPL models, which is one of the contributions of this paper.

4.1 Behavioural Ambiguities

Recall from Definition 4 that all states of a (product) LTS of an FTS are reachable from the initial state.

Definition 5 (dead transition) We say that a transition (of an FTS) is *dead* to mean that in all the FTS's products the corresponding (LTS) transition is not reachable.

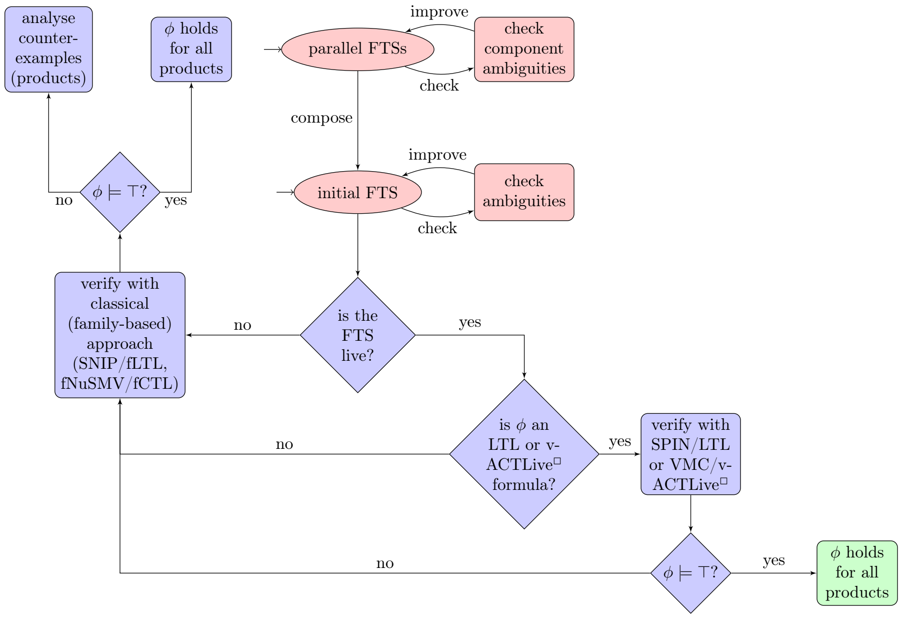

analyse counter-examples (products)
ϕ holds for all products
parallel FTSs
improve
check
check component ambiguities
compose
initial FTS
improve
check
check ambiguities
$\phi \models \top$?
no
yes
verify with classical (family-based) approach (SNIP/fLTL, fNuSMV/fCTL)
is the FTS live?
no
yes
is ϕ an LTL or v-ACTLive$^{\square}$ formula?
no
yes
verify with SPIN/LTL or VMC/v-ACTLive$^{\square}$
$\phi \models \top$?
no
yes
ϕ holds for all products

Fig. 4 Engineering methodology (top-right red part) and verification options (not in red)

Clearly, since an FTS is intended to compactly represent the behaviour of all products of a product line, a dead transition in an FTS indicates a modelling error that must be signalled to the modeller so it can be corrected. Such correction can mean removing the transition or changing its feature expression.

Definition 6 (false optional transition) We say that a transition (of an FTS) is *false optional* to mean that: (i) it is not dead, (ii) it is not annotated with the feature expression $\top$, and (iii) its corresponding (LTS) transition is present in all the FTS's products in which its source state is present.

Definition 6 is a slightly revised version of that of [9, Def. 3.2], in which condition (i) was not explicitly required. Note that condition (iii) does not imply condition (i). In fact, condition (i) requires the source state of the considered transition to be present (i.e., reachable) in at least one product of the FTS, which is not guaranteed by condition (iii).

A false optional transition in an FTS indicates a redundancy, in the sense that the associated feature expression can be replaced by $\top$ without changing the behaviour of any of the products of the product line. This redundancy may be intentional syntactic sugar, to underline the fact that the considered transition is part of the behaviour of those product configurations that satisfy the feature expression, but otherwise it may be useful for the modeller to know. Moreover, as we will see in Section 7, substitution of the feature expression with $\top$ allows for more efficient verification because it results in one more must transition, and thus one less feature expression to be evaluated.

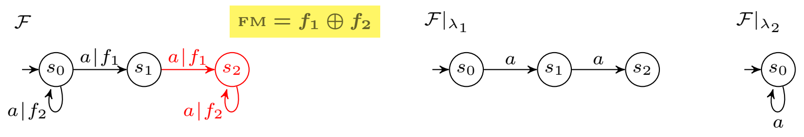


Fig. 5 FTS $\mathcal{F}$ and its product LTSs $\mathcal{F}|_{\lambda_1}$ and $\mathcal{F}|_{\lambda_2}$

Example 3 In Figure 5(left), we depict an FTS $\mathcal{F}$ with features f_1 and f_2 and feature model $\textsc{fm} = f_1 \oplus f_2$. The LTSs $\mathcal{F}|_{\lambda_1}$ and $\mathcal{F}|_{\lambda_2}$, depicted in Figure 5(middle and right), model the behaviour of its two valid product configurations: $\lambda_1 = \{f_1\}$ and $\lambda_2 = \{f_2\}$. We immediately see that transition $s_2 \xrightarrow{a \,|\, f_2} s_2$ is dead and transition $s_1 \xrightarrow{a \,|\, f_1} s_2$ is false optional.

An important safety property of systems concerns *deadlock freedom*, i.e., the system should not reach a state in which no further action is possible, thus guaranteeing progress or liveness [1,83]. In case of configurable systems (like FTSs) this notion can be extended to guaranteeing liveness for each product variant (LTS). In order to express this notion in the context of FTSs, we introduce the following definition (recall from Section 3 that a state of an FTS is said to be a deadlock if it has no outgoing transitions).

Definition 7 (hidden deadlock state) We say that a state (of an FTS) is a *hidden deadlock* to mean that: (i) it is not a deadlock in the FTS, whereas (ii) it is a deadlock in at least one of the FTS's products (LTSs).

Note that, because of condition (ii) in Definition 7, hidden deadlock states of an FTS are present in one or more of its (product) LTSs.

A hidden deadlock in the FTS should definitely be signalled to the modeller, so it can be checked whether the deadlocks in the LTSs should be remedied. If they should not, i.e., if the deadlocks in the LTSs are intended or unavoidable, then this should be made explicit in the FTS to improve understanding. Moreover, as we will see below, this enables a kind of family-based verification.

Definition 8 (ambiguous FTS) We say that an FTS is *ambiguous* to mean that: (i) at least one of its states is a hidden deadlock, or (ii) at least one of its transitions is dead or false optional.

Example 4 It is easy to see that state s_2 of the FTS $\mathcal{F}$ depicted in Figure 5(left) is a hidden deadlock state, because s_2 is a deadlock in the LTS $\mathcal{F}|_{\lambda_1}$. Indeed, $\mathcal{F}$ is an ambiguous FTS (cf. also Example 3).

Now consider the ambiguous FTS $\mathcal{F}'$ depicted in Figure 6(left) with features f_1 and f_2 and feature model $\textsc{fm} = f_1 \oplus f_2$. The LTSs $\mathcal{F}'|_{\lambda_1}$ and $\mathcal{F}'|_{\lambda_2}$, depicted in Figure 6(middle and right), model the behaviour of its two valid product configurations: $\lambda_1 = \{f_1\}$ and $\lambda_2 = \{f_2\}$. Similar to Example 3, transition $s_2 \xrightarrow{a \,|\, f_2} s_2$ is dead. However, transition $s_1 \xrightarrow{a \,|\, f_1} s_2$ is no longer false optional, since it is indeed not present in $\mathcal{F}'|_{\lambda_2}$ even though its source state s_1 is reachable in that LTS. Moreover, not only state s_2 is a hidden deadlock (for

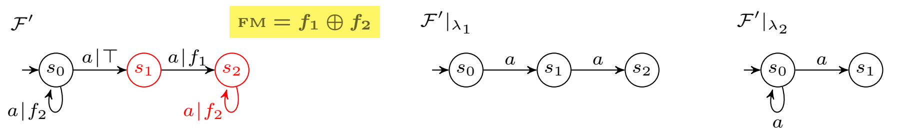


Fig. 6 FTS $\mathcal{F}'$ and its product LTSs $\mathcal{F}'|_{\lambda_1}$ and $\mathcal{F}'|_{\lambda_2}$

the same reason as above) but so is state s_1, since it is a deadlock in $\mathcal{F}'|_{\lambda_2}$. Hence also $\mathcal{F}'$ is ambiguous.

In Definition 8, an FTS is said to be ambiguous if it has a hidden deadlock state or a dead or false optional transition. We can imagine further ambiguities. For instance, consider for a moment an FTS with the two 'nearly' duplicate transitions $q \xrightarrow{a \mid f} q'$ and $q \xrightarrow{a \mid f \wedge g} q'$ (such FTSs are generally not considered in this paper, cf. Section 3). Then the second transition is redundant, since the validity of its feature expression implies that of the first transition, meaning that the second transition adds no behaviour. This clearly represents a kind of ambiguity, since looking at that second transition in isolation it would seem that execution of a requires the presence of features f and g, while actually the presence of f suffices.

Clearly, it is unlikely that systems over a certain size are modelled as single monolithic FTSs. Typically, (large) systems are designed in a modular way, as a composition of (smaller) components. We will see examples of such systems in Section 6. Our engineering methodology goes into that direction. The feedback that our analysis provides to the modellers offers them a means to revise their (small) models before composing these models to form (larger) systems.

4.2 Resolving Ambiguities

The initial part of our engineering methodology (i.e., the top-right red part of Fig. 4) concerns checking for ambiguities. Next to providing feedback to the modeller, it is important to know how to resolve ambiguities in an FTS. A dead transition could simply be removed, but this might not be the right thing to do, since the modeller may simply have made a mistake in the behavioural model or in the feature model. Likewise for a false optional transition, which however could also be intentional, to make explicit that the (corresponding) transition is part of the behaviour of those product configurations that satisfy the associated feature expression. Finally, a hidden deadlock should either be made explicit in the FTS, which can be done by adding a deadlock state to the FTS, or the deadlocks in the LTSs should be remedied—again by changing the behavioural model or the feature model. Hence, based on the detailed feedback obtained, the modeller can iteratively improve and check the FTS until the FTS is either unambiguous or ambiguous, but such that it is the FTS as intended by the modeller.

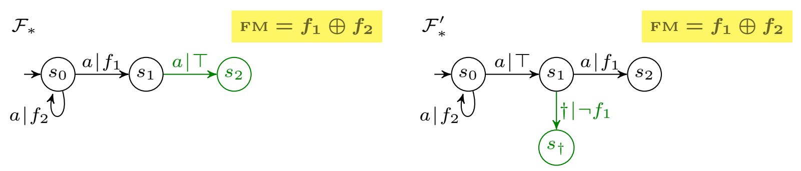


Fig. 7 Unambiguous FTSs obtained from the FTSs of Figures 5 and 6

In the latter case, according to the above recipes, any ambiguous FTS can be straightforwardly turned into an unambiguous FTS by the following transformation:

1. remove the dead transitions;
2. turn the false optional transitions into must transitions; and
3. make explicit the hidden deadlocks by adding to the set of states S of the FTS a distinguished deadlock state $s_\dagger \notin S$ and, for each hidden deadlock state s, adding a new transition (which we call a *deadlock transition*) with s as source, $s_\dagger$ as target, and labelled by a distinguished action $\dagger \notin \Sigma$ and by a feature expression that negates the disjunction of the feature expressions of all its source state's outgoing transitions.

Note that step (3) needs to be performed only for those hidden deadlock states that have not yet become explicit deadlock states upon the removal of dead transitions in step (1).

Example 5 In Figure 7(left), we depict an unambiguous FTS $\mathcal{F}_*$ that was obtained by transforming the ambiguous FTS $\mathcal{F}$ of Figure 5. We removed dead transition $s_2 \xrightarrow{a \,|\, f_2} s_2$ and false optional transition $s_1 \xrightarrow{a \,|\, f_1} s_2$ was turned into must transition $s_1 \xrightarrow{a \,|\, \top} s_2$. Note that in this case there was no need to add a deadlock transition from the hidden deadlock state s_2 to a newly added explicit deadlock state, since s_2 has become an explicit deadlock state in the FTS upon removal of the dead transition $s_2 \xrightarrow{a \,|\, f_2} s_2$.

In Figure 7(right), we depict an unambiguous FTS $\mathcal{F}'_*$ that was obtained by transforming the ambiguous FTS $\mathcal{F}'$ of Figure 6 as follows. We removed the dead transition $s_2 \xrightarrow{a \,|\, f_2} s_2$ and we added the deadlock transition $s_1 \xrightarrow{\dagger \,|\, \neg f_1} s_\dagger$ from the hidden deadlock state s_1 to the newly added explicit deadlock state $s_\dagger$. Note that in this case, without adding this deadlock transition, state s_1 would have remained a hidden deadlock state in $\mathcal{F}'_*$.

Note that the addition of explicit deadlock states and transitions does not preserve bisimilarity (nor trace equivalence), which means that resolving the ambiguities does not guarantee that the properties of the original FTS are maintained.[2] However, if a modeller decides to resolve ambiguities in an FTS (as signalled by our static analysis) through the introduction of explicit deadlock states and transitions, then even though the resulting FTS is no longer

[2] A property can still be verified by minor modifications of the formula (e.g., by expressing the v-ACTLive$^\square$ formula $EF\ [\neg a]\ \top$ as $EF\ [\neg a \wedge \neg\dagger]\ \top$ or the LTL formula $\bigcirc \top$ as $\bigcirc \neg\dagger$).

bisimilar to the original one, it has gained in clarity. Furthermore, as anticipated earlier, a kind of family-based verification on the improved FTS becomes available to the modeller, according to the strategy outlined in Figure 4.

As said before, an ambiguous FTS may be due to a mistake of the modeller in defining the feature model, in particular in the case of large feature models with many cross-tree constraints. Here we provide a small example, leaving more meaningful examples to Section 6.

Example 6 Consider again the FTS $\mathcal{F}$ depicted in Figure 5(left), but now with feature model $\textsc{fm} = f_1 \rightarrow f_2$, i.e. the presence of feature f_1 requires that of f_2. In this case, the LTS $\mathcal{F}|_{\lambda_1}$ has two further a-transitions, viz. loops in states s_0 and s_2, meaning that $\mathcal{F}$ no longer exhibits neither dead transitions nor hidden deadlock states—only the false optional transition $s_1 \xrightarrow{a \mid f_1} s_2$ remains (cf. Examples 3 and 4).

5 Detecting Ambiguities

In this section, we present an algorithm to detect behavioural ambiguity. It relies on expressing the conditions of being a hidden deadlock state, a dead transition, or a false optional transition in an FTS as propositional formulas (in which the names of the FTS's features, states and transitions are used as propositional variables), thus reducing FTS ambiguity detection to solving a set of SAT problems [41] (i.e., to decide whether a given propositional formula is satisfiable). While SAT solving is well known to be NP-complete, SAT solvers are widely used for all kinds of static analysis on feature models with a surprising effectiveness even for models with hundreds of thousands of clauses and tens of thousands of variables [84,79].

To this aim, our implementation exploits an automatic SAT solver. SAT solving is an active field of research [70,29,72,6] and tools exist that compute, more or less efficiently, a solution for an input formula, or fail if the formula is not satisfiable. Hence, by feeding the formula encoding an ambiguity question to a SAT solver, we can obtain an answer to it. In our implementation, we use the Z3 SMT solver [85] (that includes a SAT solver) developed by Microsoft Research and freely available under the MIT license. The python code of our implementation is publicly available [10]; it accepts FTSs in the format .dot as input and all example models used in the remainder of this paper are provided.

5.1 FTS representation

Our algorithm assumes that the considered FTS is represented by the global data structure `fts` that includes four fields:

1. `states` stores the set of all states in the FTS;
2. `transitions` stores the set of all transitions in the FTS;
3. `initial` stores the initial state of the FTS;

4. `fm` stores the formula FM (introduced before Example 2 in Section 3), which is a formula in $\mathbb{B}(F)$ that represents the feature model of the FTS.

Each state is represented by a data structure that includes three fields:

1. `in_trs` stores the set of incoming transitions of this state;
2. `out_trs` stores the set of outgoing transitions of this state;
3. `hdead` is a Boolean flag used to record whether this state is a hidden deadlock.

Each transition is represented by a data structure that includes four fields:

1. `bx` stores the feature expression labelling the transition, i.e., a propositional formula in $\mathbb{B}(F)$;
2. `source` stores the source state of the transition;
3. `dead` is a Boolean flag used to record whether this transitions is dead;
4. `false_opt` is a Boolean flag used to record whether this transitions is false optional.

The Boolean flags in each state (field `hdead`) and transition (fields `dead` and `false_opt`) are used to record the results of the analysis (i.e., the output of the algorithm); their initial values are immaterial.

5.2 Propositional Formulas Expressing the Conditions to be Checked

Let $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$ be an FTS. Let T be the set of the names of the transitions of the FTS. In this section, we introduce propositional formulas on $\mathbb{B}(F \cup S \cup T)$ that express the conditions of being a hidden deadlock state, a dead transition, or a false optional transition in the FTS.

Recall that an interpretation for a propositional formula in $\mathbb{B}(F \cup S \cup T)$ is a function $\mathcal{I} : (F \cup S \cup T) \longrightarrow \{\top, \bot\}$. We say that a state or transition is *selected* in an interpretation to mean that the associated propositional variable gets value $\top$ and, on the other hand, we say it is *deselected* in an interpretation to mean that the associated propositional variable gets value $\bot$.

Notation 1 *For the sake of simplicity, we abuse the notation of data structures for states and transitions (cf. Section 5.1). We use `fts.states` as an alternative name for S, and use `fts.transitions` as an alternative name for T. We use `fts.initial` to refer to the initial state s_0, use $s \in$ `fts.states` and $s =$ `t.source` (where $t \in T$) to refer to the corresponding state (an element of S), and use $t \in$ `fts.transitions`, $t \in$ `s.in_trs`, and $t \in$ `s.out_trs` to refer to the corresponding transition (an element of T).*

Let `inner_states` denote the set `fts.states` $\setminus$ {`fts.initial`}. An *initial path* is a path that starts from the initial state.

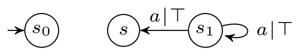
We first introduce some propositional formulas that, together with the formula `fts.fm`, allow us to formalise the conditions that grasp the initial paths in the FTS's products.

- $\phi_{\texttt{initial}}$ is the formula $\texttt{fts.initial}$ (i.e., the name of the initial state). This formula is valid in an interpretation $\mathcal{I}$ iff $\mathcal{I}$ selects the initial state.
- $\phi_{\texttt{inner}}$ is the formula $\bigwedge_{s \in \texttt{inner\_states}} (s \Rightarrow \texttt{atLeastOneTransitionOf}(s.\texttt{in\_trs}))$, where $\texttt{atLeastOneTransitionOf}(X)$ is a placeholder for $\bigvee_{t \in X}(t.\texttt{bx} \wedge t \wedge t.\texttt{source})$. This formula is valid in an interpretation $\mathcal{I}$ iff $\mathcal{I}$ selects only states that are reachable via selected transitions, with valid (in $\mathcal{I}$) feature expressions, that are outgoing from selected states.
- $\phi_{\texttt{single}}$ is the formula $\bigwedge_{s \in \texttt{fts.states}} \texttt{atMostOneOf}(s.\texttt{out\_trs})$, where $\texttt{atMostOneOf(X)}$ is a placeholder for $\bigwedge_{t \in X} t \Rightarrow (\bigwedge_{t' \in X \setminus \{t\}} \neg t')$. This formula is valid in an interpretation $\mathcal{I}$ iff $\mathcal{I}$ selects at most one outgoing transition, for each state (selected or not).
- $\texttt{end}(s)$ is the formula $s \wedge (\bigwedge_{t \in s.\texttt{out\_trs}} \neg t)$. This formula is valid in an interpretation $\mathcal{I}$ iff $\mathcal{I}$ selects the state s and deselects all outgoing transitions from that state.

Next, we focus our attention on the conjunction of the above formulas.

- $\texttt{is\_useful\_state}(s)$ is the formula $\texttt{fts.fm} \wedge \phi_{\texttt{initial}} \wedge \phi_{\texttt{inner}} \wedge \phi_{\texttt{single}} \wedge \texttt{end}(s)$. This formula is satisfiable (i.e., valid in some interpretation $\mathcal{I}$) iff in at least one LTS product there is a simple path (i.e., a path with no repeated states) that starts from the initial state and ends in s.[3]

Example 7 Consider the FTS on the right.



It has no features and just one product configuration (represented by the mapping from the empty set to $\mathbb{B}$) which yields the LTS consisting of the initial state s_0. Therefore, $\texttt{fts.fm} = \top$. States s and s_1 are not useful (since they are not reachable from s_0) and, accordingly, the formulas $\texttt{is\_useful\_state}(s)$ and $\texttt{is\_useful\_state}(s_1)$ are not satisfiable. To see this, let t be the transition from s_1 to s and let t_1 be the transition from s_1 to s_1.

- To satisfy $\texttt{is\_useful\_state}(s)$ requires to assign $\top$ to s (because $\texttt{end}(s)$ must be satisfied), which in turn requires to assign $\top$ to both t and s_1 (because $\phi_{\texttt{inner}}$ must be satisfied), which in turn requires to assign $\bot$ to t_1 (because $\phi_{\texttt{single}}$ must be satisfied, viz. only t can exit s_1), which in turn implies that $\phi_{\texttt{inner}}$ cannot be satisfied (because at least one transition has to enter s_1) and therefore $\texttt{is\_useful\_state}(s)$ cannot be satisfied.
- To satisfy $\texttt{is\_useful\_state}(s_1)$ requires to assign $\top$ to s_1 and $\bot$ to t_1 (because $\texttt{end}(s_1)$ must be satisfied), which in turn implies that $\phi_{\texttt{inner}}$ (and therefore $\texttt{is\_useful\_state}(s_1)$) cannot be satisfied.

We can straightforwardly define the formulas for checking the behavioural ambiguities by exploiting the formula $\texttt{is\_useful\_state}(s)$.

- $\texttt{exists\_deadlock}(s)$ is the formula $\texttt{is\_useful\_state}(s) \wedge \bigwedge_{t \in s.\texttt{out\_trs}} \neg t.\texttt{bx}$. This formula is satisfiable iff, in at least one LTS product, the state s is a

[3] Note that there could be interpretations that fulfill $\texttt{is\_useful\_state}(s)$ and include also non-initial paths, but in any case s must still be reachable by an initial path that is within the interpretation.

deadlock—thus if s is not a deadlock in the FTS, then s is a hidden deadlock (cf. Definition 7).

– `is_not_dead_transition`(t) is the formula `is_useful_state(`t`.source)` $\wedge$ t`.bx`. This formula is satisfiable iff the transition t is not dead (cf. Definition 5).
– `may_be_opt_transition`(t) is the formula `is_useful_state(`t`.source)` $\wedge \neg t$`.bx`. This formula is satisfiable iff the LTS transition corresponding to transition t (of the FTS) is not present in at least one of the FTS's products in which its source state is present—thus if t is not dead, then t is not false optional (cf. Definition 6).

Example 8 Consider the FTS $\mathcal{F}$ of Example 3. Let t_0, t_1, t_2, and t_3 be the transitions $s_0 \xrightarrow{a \mid f_2} s_0$, $s_0 \xrightarrow{a \mid f_1} s_1$, $s_1 \xrightarrow{a \mid f_1} s_2$, and $s_2 \xrightarrow{a \mid f_2} s_2$, respectively. We have that the formula `is_not_dead_transition`(t_3) is not satisfiable (therefore t_3 is dead) and the formula `exists_deadlock`(s_2) is satisfiable (therefore state s_2 is a hidden deadlock). Moreover, the formula `is_not_dead_transition`(t_2) is satisfiable (therefore t_2 is not dead) and the formula `may_be_opt_transition`(t_2) is satisfiable (therefore t_2 is false optional).

Consider the FTS $\mathcal{F}'$ of Example 4. Let t_0, t_1, t_2, and t_3 be the transitions $s_0 \xrightarrow{a \mid f_2} s_0$, $s_0 \xrightarrow{a \mid \top} s_1$, $s_1 \xrightarrow{a \mid f_1} s_2$, and $s_2 \xrightarrow{a \mid f_2} s_2$, respectively. We have that the formula `is_not_dead_transition`(t_3) is not satisfiable (therefore t_3 is dead). Moreover, both formulas `exists_deadlock`(s_1) and `exists_deadlock`(s_2) are satisfiable (therefore both s_1 and s_2 are hidden deadlocks).

We denote by $\lambda_{\mathcal{I}}$ the restriction of the interpretation $\mathcal{I}$ to features. The following lemma formally states the meaning of the five components of the formula `is_useful_state`(s).

Lemma 1 *Let* `fts` *be the global data structure that represents the FTS* $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$ *and let* $\mathcal{I}$ *be an interpretation. Then*

1. $\mathcal{I} \models$ `fts.fm` *iff* $\lambda_{\mathcal{I}} \in \Lambda$.
2. $\mathcal{I} \models \phi_{initial}$ *iff* $\mathcal{I}($`fts.initial`$) = \top$.
3. $\mathcal{I} \models \phi_{inner}$ *iff, for all* $s \in$ `inner_states`*, if* $\mathcal{I}(s) = \top$ *then there is at least a transition* $t \in s$`.in_trs` *such that:* $\mathcal{I} \models t$`.bx`*,* $\mathcal{I}(t) = \top$*, and* $\mathcal{I}(t$`.source`$) = \top$.
4. $\mathcal{I} \models \phi_{single}$ *iff, for all* $s \in$ `inner_states`*, there is at most one transition* $t \in s$`.out_trs` *such that* $\mathcal{I}(t) = \top$.
5. $\mathcal{I} \models$ `end`(s) *iff,* $\mathcal{I}(s) = \top$ *and* $\mathcal{I}(t) = \bot$*, for all* $t \in s$`.out_trs`*, where* $s \in$ `fts.states`.

Proof Straightforward. □

The next lemma formally states the meaning of the formula `is_useful_state`(s).

Lemma 2 *Let* `fts` *be the global data structure that represents the FTS* $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$ *and let* s' *be a state of* $\mathcal{F}$*. Then the formula* `is_useful_state`(s') *is satisfiable iff there are an interpretation* $\mathcal{I}$ *and an initial path* P *of* $\mathcal{F}$ *ending in* s' *such that* $\lambda_{\mathcal{I}} \in \Lambda$ *and*

1. $\mathcal{I}(s) = \top$*, for each state* s *visited by* P*, and*

2. $\mathcal{I}(t) = \top$ *and* $\lambda_{\mathcal{I}} \models t.\texttt{bx}$*, for each transition* t *visited by* P.

Proof We consider first the direction from right to left. Let $\mathcal{I}$ be an interpretation and P an initial path of $\mathcal{F}$ ending in s' such that $\lambda_{\mathcal{I}} \in \Lambda$ and conditions (1) and (2) hold. Consider the interpretation $\mathcal{I}_0$ that maps to $\bot$ all the states and transitions that are not in P and behaves as $\mathcal{I}$ on all other arguments. Then it is immediate to check that $\mathcal{I}_0 \models \texttt{is\_useful\_state}(s')$ holds (i.e., $\texttt{is\_useful\_state}(s')$ is satisfiable).

Consider now the other direction. Let $\mathcal{I}'$ be an interpretation satisfying $\texttt{is\_useful\_state}(s')$, i.e., such that $\mathcal{I}' \models \texttt{fts.fm}$, $\mathcal{I}' \models \phi_{\texttt{initial}}$, $\mathcal{I}' \models \phi_{\texttt{inner}}$, $\mathcal{I}' \models \phi_{\texttt{single}}$, and $\mathcal{I}' \models \texttt{end}(s')$ hold.

Immediately, $\lambda_{\mathcal{I}'} \in \Lambda$ follows from $\mathcal{I}' \models \texttt{fts.fm}$, while $\mathcal{I}'(\texttt{fts.initial}) = \top$ and $\mathcal{I}'(s') = \top$ follow from $\mathcal{I}' \models \phi_{\texttt{initial}}$ and $\mathcal{I}' \models \texttt{end}(s')$, respectively. Then the proof follows by induction on the number n of states selected by $\mathcal{I}'$ (note that n must be at least one, since $\texttt{fts.initial}$ is always selected).

– If $n = 1$, then we are selecting only one state, i.e., s' and the initial state coincide. Hence, the initial path is just s' and the proof that conditions (1) and (2) hold is immediate.
– Let $n > 1$. If s' is the initial state, then the proof is immediate (as for the case $n = 1$). Thus, let s' be different from the initial state. We know that s' is selected in whatever interpretation satisfying $\texttt{is\_useful\_state}(s')$. By Lemma 1(3), we know that there are $m \geq 1$ transitions $\{t_1, \ldots, t_m\} \in s.\texttt{in\_trs}$ such that $\mathcal{I}' \models t_i.\texttt{bx}$, $\mathcal{I}'(t_i) = \top$, and $\mathcal{I}'(t_i.\texttt{source}) = \top$. Moreover, we know that $s'.\texttt{out\_trs} = \varnothing$ by Lemma 1(5).

  Let $\mathcal{I}_0$ be the interpretation that maps $\{s', t_1, \ldots, t_m\}$ in $\bot$ and behaves as $\mathcal{I}'$ on all other arguments. For all transitions t_i, we have that $\mathcal{I}_0 \models \texttt{end}(t_i.\texttt{source})$ holds, because by Lemma 1(3) there is at most a selected transition outgoing from $t_i.\texttt{source}$ in $\mathcal{I}'$ and we deselected it. Moreover, $\mathcal{I}_0 \models \texttt{fts.fm}$, $\mathcal{I}_0 \models \phi_{\texttt{initial}}$, $\mathcal{I}_0 \models \phi_{\texttt{inner}}$, and $\mathcal{I}_0 \models \phi_{\texttt{single}}$ hold. Therefore $\mathcal{I}_0 \models \texttt{is\_useful\_state}(t_i.\texttt{source})$ holds.

  By induction we have that there are a configuration $\lambda_{\mathcal{I}_0} \in \Lambda$ and a selected initial path P_0 of $\mathcal{F}|_{\lambda_{\mathcal{I}_0}}$ that reaches $t_1.\texttt{source}$ and (together with $\mathcal{I}_0$) satisfies conditions (1) and (2). Clearly, $\lambda_{\mathcal{I}_0} = \lambda_{\mathcal{I}'}$ (by construction of $\mathcal{I}_0$). Extending P_0 with the transition t_1 and the state s', we obtain an initial path P' that reaches s' and (together with $\mathcal{I}'$) satisfies conditions (1) and (2). □

Finally, the following theorem formally states the correctness of the formulas $\texttt{exists\_deadlock}(s)$, $\texttt{is\_not\_dead\_transition}(t)$, and $\texttt{may\_be\_opt\_transition}(t)$.

Theorem 1 (correctness of the formulas for checking the behavioural ambiguities) *Let* $\textit{fts}$ *be the global data structure representing the FTS* $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$ *and let* s *be a state of* $\mathcal{F}$*. Then*

1. The formula $\texttt{exists\_deadlock}(s)$ *is satisfiable iff there is a configuration* $\lambda \in \Lambda$ *such that the state* s *is a deadlock in* $\mathcal{F}|_{\lambda}$.

2. *The formula* `is_not_dead_transition`(t) *is satisfiable iff there is a configuration* $\lambda \in \Lambda$ *such that the LTS transition corresponding to transition* t *is reachable in* $\mathcal{F}|_\lambda$.
3. *The formula* `may_be_opt_transition`(t) *is satisfiable iff there is a configuration* $\lambda \in \Lambda$ *such that the state* t`.source` *is reachable in* $\mathcal{F}|_\lambda$ *and the LTS transition corresponding to transition* t *is not reachable in* $\mathcal{F}|_\lambda$.

Proof Straightforward from Lemma 2. □

5.3 Algorithms

The algorithm in Listing 1 below uses the function `check` to verify whether a propositional formula ϕ is satisfiable, namely to verify the existence of an interpretation (an assignment of truth values to propositions in $\mathbb{B}(F \cup S \cup T)$) that makes the formula valid. This is the core functionality of all SAT solvers.

Listing 1 Ambiguities discovery algorithm

```
# fts contains the input FTS (according to Section 5.1)

for $s$ in fts.states:
   if($s$.out_trs $= \emptyset$):
      $s$.hdead $\leftarrow$False
   else:
      $s$.hdead $\leftarrow$check(exist_deadlock($s$))

# for all states s, it holds that:
# (s.hdead $\equiv$ ``s is a hidden deadlock'')

for $s$ in fts.states:
   for $t$ in $s$.in_trs:
      $t$.dead $\leftarrow$not check(is_not_dead_transition($t$))
      if($t$.dead or $t$.bx $= \top$):
         $t$.false_opt $\leftarrow$False
      else:
         $t$.false_opt $\leftarrow$not check(may_be_opt_transition($t$))

# for all transitions t, it holds that:
# (t.dead $\equiv$ ``t is dead'') and (t.false_opt $\equiv$ ``t is false optional'')
```

Theorem 2 (correctness of the ambiguities discovery algorithm) *Let* `fts` *be a data structure representing an FTS. The execution of the algorithm in Listing 1 terminates and at the end of the execution the following holds.*

1. *For each state* s, *if* s *is a hidden deadlock, then* s`.hdead=`**True**; *otherwise* s`.hdead=`**False**.
2. *For each transition* t,
   (a) *if* t *is dead, then* t`.dead=`**True**; *otherwise* t`.dead=`**False**;
   (b) *if* t *is false optional, then* t`.false_opt=`**True**; *otherwise* t`.false_opt=`**False**.

Proof Correctness of the formulas `exists_deadlock`(s), `is_not_dead_transition`(t), and `may_be_opt_transition`(t) is stated by Theorem 1.

The algorithm first detects all the hidden deadlocks (lines 3–7, where the test in line 4 detects the states that are deadlocks in the FTS), thus establishing the invariant in lines 9–10. Then it detects all the dead transitions and all the false optional transitions (lines 12–18, where the test in line 15 detects the dead transitions that cannot be false optional because they are dead or labelled with $\top$), thus establishing the invariant in lines 20–21 while keeping the invariant in lines 9–10 (since the Boolean flags `hdead` of the states are not modified).

The termination of the algorithm is straightforward since the number of states and transitions of the FTS is finite. $\square$

It is worth observing that, whenever one is only interested in detecting the hidden deadlocks, it is enough to run only the first part (lines 1–10) of the algorithm in Listing 1: this part represents a specialised algorithm that only detects hidden deadlocks.

Remark 1 (The FTS ambiguity detection problem is NP-complete) For every propositional formula ϕ with variables in F, the FTS

$$(\{s_0, s\}, \{a\}, s_0, \{s_0 \xrightarrow{a \mid \phi} s\}, F, 2^F)$$

is such that its (unique) transition is: (i) dead if and only if ϕ is not satisfiable (i.e. $\neg\phi$ is valid); and (ii) false optional if and only if $\neg\phi$ is not satisfiable (i.e. ϕ is valid). Moreover, state s_0 is a hidden deadlock if and only if ϕ is not satisfiable. Thus, the FTS ambiguity detection problem is NP-hard. Moreover, the algorithm in Listing 1 can be transformed into an algorithm that, given the data structure `fts` (cf. Section 5.1) representing an FTS $\mathcal{F}$ with n states and m transitions, reduces (in polynomial time in the size of `fts`) the ambiguity detection problem for $\mathcal{F}$ to $n + 2 \times m$ SAT problems (each problem consisting of a formula whose size is linear in the size of `fts`), as follows:

– extend the data structure introduced in Section 5.1 by adding a field `hdead_formula` to each state and by adding a field `dead_formula` and a field `false_optional_formula` to each transition;
– replace line 7 by s.`hdead_formula` $\leftarrow$`exist_deadlock(`s`)`;
– replace line 14 by t.`dead_formula` $\leftarrow$`is_not_dead_transition(`s`)`; and
– replace lines 15–18 by t.`false_opt_formula` $\leftarrow$`may_be_opt_transition(`s`)`.

Solving the SAT problems stored in the fields `hdead_formula`, `dead_formula`, and `false_optional_formula` provides a solution to the ambiguity detection problem for the given FTS, therefore we conclude that the FTS ambiguity detection problem is NP-complete.

6 Benchmark Examples

In this section, we apply the new algorithm to a number of exemplary FTSs from the literature. The python code of the implementation and all FTS models

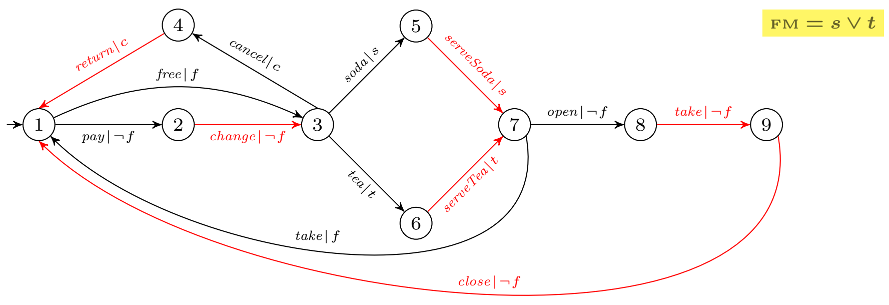


Fig. 8 FTS of the vending machine from [36]

allowing the verification of the examples presented in this section are publicly available [10]. We first discuss the experiments (in Section 6.1) and then the corresponding performance results (in Section 6.2).

6.1 Experiments

Vending Machine In Figure 8, we depict the FTS modelling the behaviour of a configurable vending machine from [36], an FTS benchmark which was used in [9] and in many other publications [40,39,51,7,12,54,60,62,32,59,63,8,57]. It serves a beverage (soda or tea) either for free or upon payment, in which case a compartment is opened for the customer to take the beverage after which it closes again. Its feature model is represented by the formula $s \lor t$ over the 4 features $\{f, c, s, t\}$, thus resulting in 12 product configurations (viz. $2^4 - 4$, excluding the product configurations $\varnothing$, $\{f\}$, $\{c\}$, and $\{f, c\}$ that lack both features s for soda and t for tea). The FTS of the vending machine contains only 9 states and 13 transitions.

Listing 2 reports the result of applying our static analysis algorithm to this FTS. The FTS contains no dead transitions and no hidden deadlocks, but it does contain the 6 false optional transitions $(2, \mathit{change}, \neg f, 3)$, $(4, \mathit{return}, c, 1)$, $(5, \mathit{serveSoda}, s, 7)$, $(6, \mathit{serveTea}, t, 7)$, $(8, \mathit{take}, \neg f, 9)$, and $(9, \mathit{close}, \neg f, 1)$. Thus, the FTS is ambiguous, but it would suffice to turn its false optional transitions into must transitions to make the FTS unambiguous.

Listing 2 Result of the static analysis on the FTS of Figure 8

```
VENDING MACHINE: live
 LIVE STATES = [1,2,3,4,5,6,7,8,9]
 DEAD TRANSITIONS = []
 FALSE OPTIONAL TRANSITIONS = [(2,3,change),(4,1,return),(5,7,serveSoda),
  (6,7,serveTea),(8,9,take),(9,1,close)]
 HIDDEN DEADLOCK STATES = []
```

Coffee Machine In Figure 9, we depict the FTS modelling the behaviour of a configurable coffee machine family from [25]. Originally introduced in [66], this is another SPL benchmark which was already used in [9] and in a number

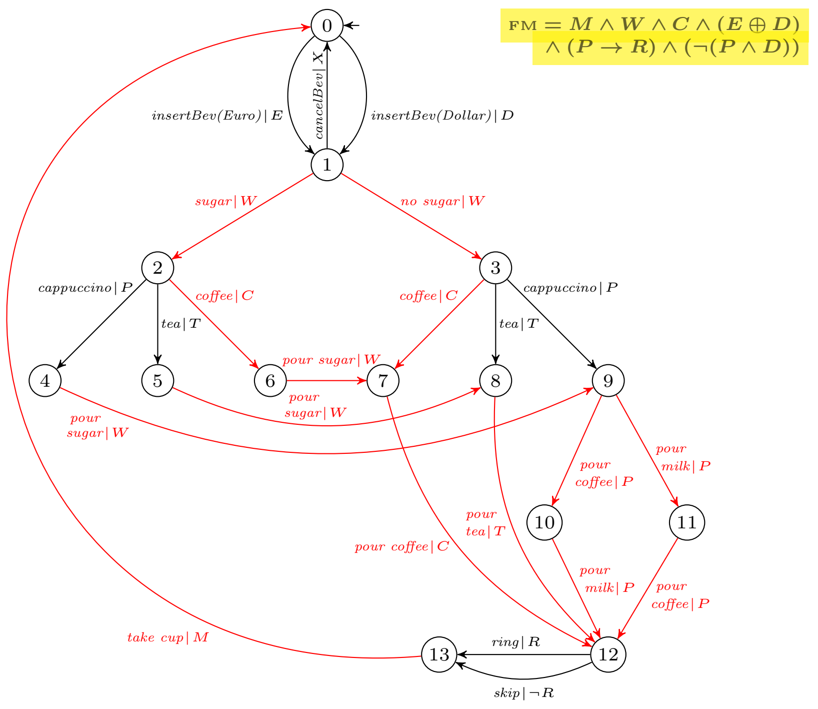


Fig. 9 FTS of the coffee machine from [25]

of other publications [4,5,15,17,20,23,22,13,21,27,91].[4] The coffee machine serves a (possibly sugared) beverage (coffee, tea, or cappuccino) upon the insertion of a coin (euro or dollar), after which the customer takes her/his beverage (possibly following a ringtone). Its feature model is represented by the formula $\text{FM}_\text{C} = M \wedge W \wedge C \wedge (E \oplus D) \wedge (P \rightarrow R) \wedge (\neg(P \wedge D))$ over the features $F_c = \{M, W, C, E, D, P, R, T, X\}$, resulting in 12 product configurations which accept either euros or dollars and offer coffee (with sugar) and possibly tea and cappuccino (upon a ringtone and only for euros). The FTS of the coffee machine contains 14 states and 23 transitions.

Listing 3 reports the result of applying our static analysis algorithm to this FTS. The FTS contains no dead transitions and no hidden deadlocks, but it contains 14 false optional transitions such as $(1, \mathit{sugar}, W, 2)$, $(1, \mathit{no\ sugar}, W, 3)$, $(2, \mathit{coffee}, C, 6)$, $(8, \mathit{pour\ tea}, T, 12)$, and $(13, \mathit{take\ cup}, M, 0)$. Thus, the FTS is ambiguous, but it would suffice to turn its false optional transitions into must transitions to make the FTS unambiguous.

Listing 3 Result of the static analysis on the FTS of Figure 9

```
COFFEE MACHINE: live
```

[4] The only differences between the FTS used here and the one in [9] is the additional transition $(1, \mathit{cancelBev}, X, 0)$, which allows to cancel a coin insertion in the presence of an additional optional feature X, next to a renaming of the states and the features.

```
 LIVE STATES = [0,1,2,3,4,5,6,7,8,9,10,11,12,13]
 DEAD TRANSITIONS = []
 FALSE OPTIONAL TRANSITIONS = [(1,2,sugar),(1,3,no_sugar),(2,6,coffee),
  (3,7,coffee),(6,7,pour_sugar),(5,8,pour_sugar),(4,9,pour_sugar),
  (9,11,pour_milk),(9,10,pour_coffee),(8,12,pour_tea),(7,12,pour_coffee),
  (11,12,pour_coffee),(10,12,pour_milk),(13,0,take_cup)]
 HIDDEN DEADLOCK STATES = []
```

Coffee/Soup Machine In [25], this family of coffee machines was extended with an optional soup component running in parallel with the beverage component. The FTS modelling the behaviour of this soup component is depicted in Figure 10.[5] The resulting family of vending machines is such that each product allows the insertion of either euros or dollars (returned upon a cancel) in one of its components. The customer chooses a beverage or, if available, a type of soup (at least one among chicken, tomato, pea), which requires to place a cup. A cup detector is optional (mandatory for dollars). Whenever present, soup is only poured if a cup was placed. Placing a cup may need to be repeated if not detected. A choice for soup may be cancelled until a cup is detected. Optionally, a ringtone may ring upon delivery (mandatory for cappuccino, as before), after which the customer takes her/his cup (with a drink or soup) and can again insert a coin in one of the components. The feature model of the soup component is represented by the formula $\text{FM}_\text{C} \wedge \text{FM}_\text{S} \wedge SC$, where $\text{FM}_\text{S} = (U \rightarrow SC) \wedge (S \leftrightarrow SC) \wedge (CS \vee PS \vee TS \vee \neg S) \wedge ((D \wedge SC) \rightarrow U)$, over the features $F_c \cup \{SC, U, S, CS, PS, TS\}$. The FTS of the soup component contains 13 states and 28 transitions.

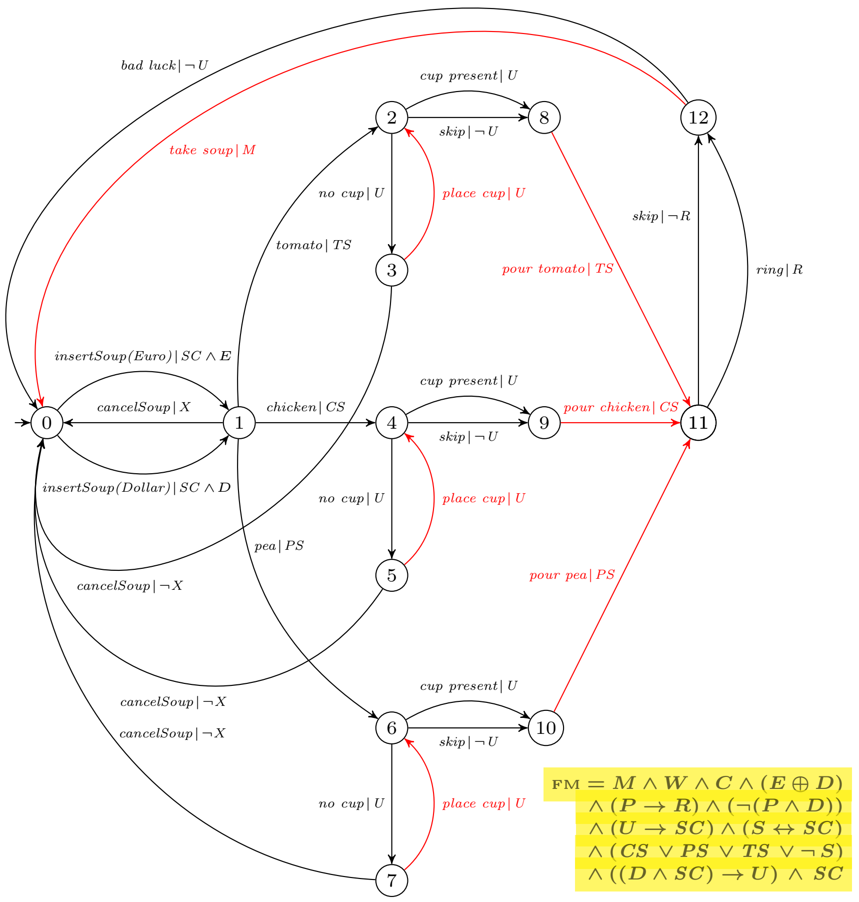
Listing 4 reports the result of applying our static analysis algorithm to this FTS. The FTS contains no hidden deadlocks and no dead transitions, but it contains the 7 false optional transitions $(3, \textit{place cup}, U, 2)$, $(5, \textit{place cup}, U, 4)$, $(7, \textit{place cup}, U, 6)$, $(8, \textit{pour tomato}, TS, 11)$, $(9, \textit{pour chicken}, CS, 11)$, $(10, \textit{pour pea}, PS, 11)$, and $(12, \textit{take soup}, M, 0)$. Thus, the FTS is ambiguous, but it would suffice to turn its false optional transitions into must transitions to make the FTS unambiguous.

Listing 4 Result of the static analysis on the FTS of Figure 10

```
SOUP COMPONENT: live
 LIVE STATES = [0,1,2,3,4,5,6,7,8,9,10,11,12]
 DEAD TRANSITIONS = []
 FALSE OPTIONAL TRANSITIONS = [(3,2,place_cup),(5,4,place_cup),(7,6,place_cup),
  (8,11,pour_tomato),(9,11,pour_chicken),(10,11,pour_pea),(12,0,take_soup)]
 HIDDEN DEADLOCK STATES = []
```

The feature model of the composite FTS that results from running the (optional) soup component depicted in Figure 10 in parallel with the beverage component of the FTS of the coffee machine depicted in Figure 9 is represented by the formula $\text{FM}_\text{C} \wedge \text{FM}_\text{S}$ over the features $F_c \cup \{SC, U, S, CS, PS, TS\}$, giving

[5] While omitted in the component FTS drawn in [25], once put in parallel, coin insertion for soup requires the presence of the soup component (e.g., $(0, \textit{insertSoup(Euro)}, SC \wedge E, 1)$.



$$
\begin{aligned}
\textsc{fm} = \boldsymbol{M \wedge W \wedge C \wedge (E \oplus D)} \\
\boldsymbol{\wedge\, (P \rightarrow R) \wedge (\neg(P \wedge D))} \\
\boldsymbol{\wedge\, (U \rightarrow SC) \wedge (S \leftrightarrow SC)} \\
\boldsymbol{\wedge\, (CS \vee PS \vee TS \vee \neg S)} \\
\boldsymbol{\wedge\, ((D \wedge SC) \rightarrow U) \wedge SC}
\end{aligned}
$$

Fig. 10 FTS of the soup component from [25]

rise to 244 product configurations.[6] The composite FTS contains 182 states and 691 transitions.

Listing 5 reports the result of applying our static analysis algorithm to this composite FTS. The composite FTS contains no hidden deadlocks, 284 false optional transitions and 8 dead transitions. The false optional transitions are obviously due to the relatively large amount of false optional transitions in the two component FTSs. The dead transitions can be explained by analysing the execution traces. Consider, for instance, the dead transition $(12, \mathit{insertSoupDollar}, SC \wedge D, 29)$. Its source state can be reached upon inserting a coin, followed by choosing sugar and ordering cappuccino, which we recall to require feature P. If the inserted coin was a euro, requiring feature E,

[6] In [25], only 118 of these configurations are valid due to additional quantitative constraints on feature attributes omitted here (e.g. cost of features). We also omitted some mandatory features that do not occur in the FTSs and are thus irrelevant for our purposes.

then the transition cannot be executed since features E and D exclude each other, while if the inserted coin was a dollar, requiring feature D, then the transition cannot be executed since P and D exclude each other. Since any product has either D or E, indeed in all product LTSs this transition is not reachable. A similar reasoning applies to the *skip* transitions, which require a feature R that cannot be part of product LTSs in which their sources states are reachable. Hence, the FTS is ambiguous, but it would suffice to remove its dead transitions and turn its false optional transitions into must transitions to make the FTS unambiguous.

Listing 5 Result of the static analysis on the composite FTS resulting from the parallel composition of the beverage component of the FTS of the coffee machine depicted in Figure 9 and the soup component depicted in Figure 10

```
COFFEE SOUP MACHINE: live
 LIVE STATES = [1,2,...,182]
 DEAD TRANSITIONS = [(12,29,insertSoupDollar),(16,38,insertSoupDollar),
  (36,72,insertSoupDollar),(37,73,insertSoupDollar),(136,165,skip),
  (161,177,skip),(175,180,skip),(176,181,skip)]
 FALSE OPTIONAL TRANSITIONS = [(2,4,sugar),(2,5,no_sugar),...,(182,64,take_soup)]
 HIDDEN DEADLOCK STATES = []
```

Since neither the beverage component nor the soup component has any dead transitions, this shows that the parallel composition of FTSs (with some features in common) without dead transitions may result in a composite FTS with dead transitions. Furthermore, the size of the composite FTS is such that analysis by hand is infeasible. In the remainder of this section, we consider even larger examples to illustrate the scalability of our approach.

Mine Pump In Figure 11, we depict the FTS modelling the behaviour of the *system* FTS modelling the logic of a configurable controller of the mine pump model from [35,36], a standard SPL benchmark for FTSs which was used in [9] and in many other publications [40,37,43,39,45,61,54,60,24,62,18]. The controller of this mine pump model is the parallel composition of the *system* FTS with the *state* FTS, depicted in Figure 12. The mine pump has to keep a mine safe from flooding by pumping water from a shaft while avoiding a methane explosion. Therefore, the controller interacts with an environment: it operates a water pump based on water and methane level sensors, modelled by three further FTSs. The parallel composition of these five FTSs constitutes the complete mine pump model. We depict the FTS of the methane level in Figure 13 and refer to [35,36] for the remaining FTSs. The feature model of the mine pump model can be represented by the formula $\phi = (c \leftrightarrow (ct \vee cp)) \wedge l$ over the feature set $F = \{c, ct, cp, m, l, ll, ln, lh\}$, thus resulting in 64 products (viz. 2^6, since ϕ is equivalent to considering features $\{ct, cp, m, ll, ln, lh\}$ to be optional). The *system* FTS of the mine pump model contains 25 states and 41 transitions.[7] The controller of the mine pump model, composed of the *system* and *state* FTSs, contains 77 states and 104 transitions. The complete mine pump model, composed of five FTSs, contains 418 states and 1,255 transitions.

[7] Transitions with more than one label are abbreviations for one transition for each label.

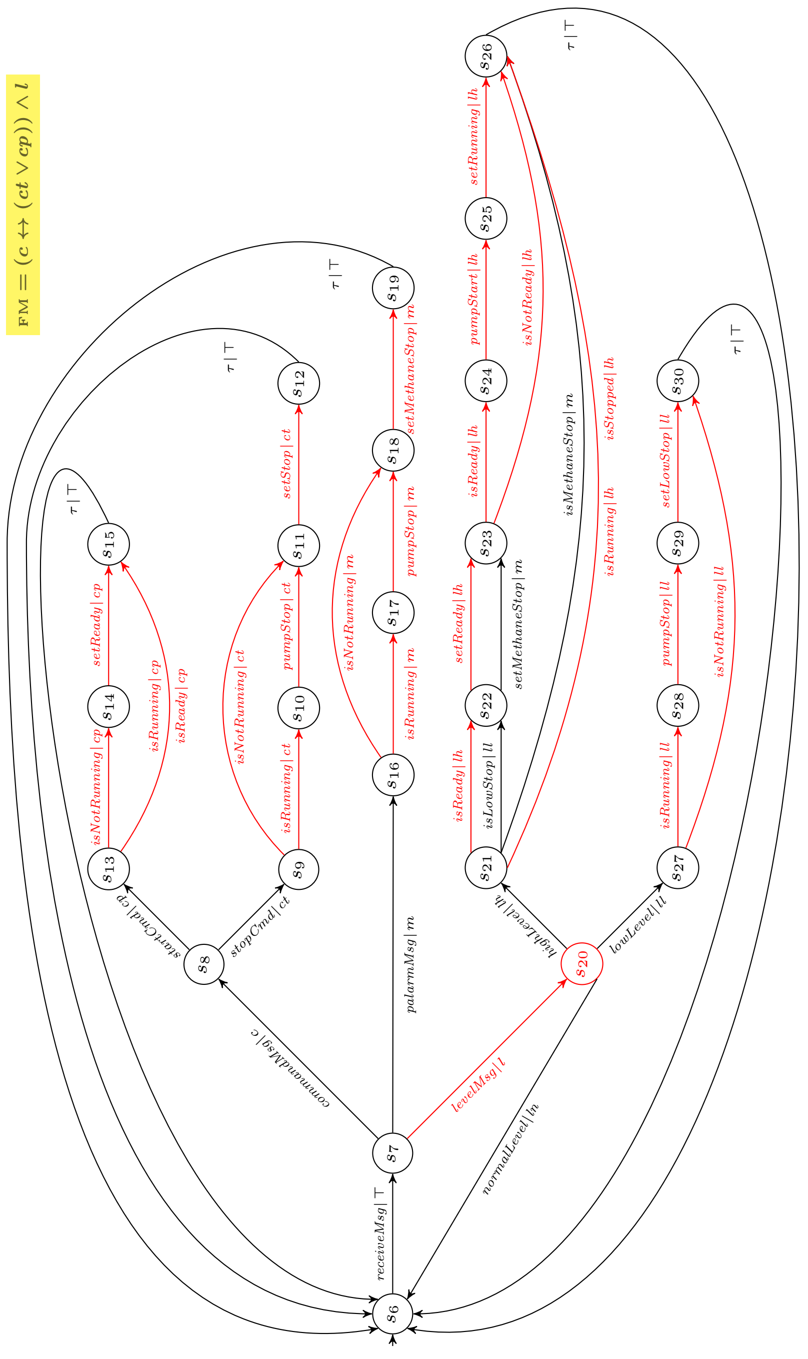


Fig. 11 The *system* FTS of the mine pump model from [36] (we have labelled transitions that were unlabelled with $\tau\,|\,\top$)

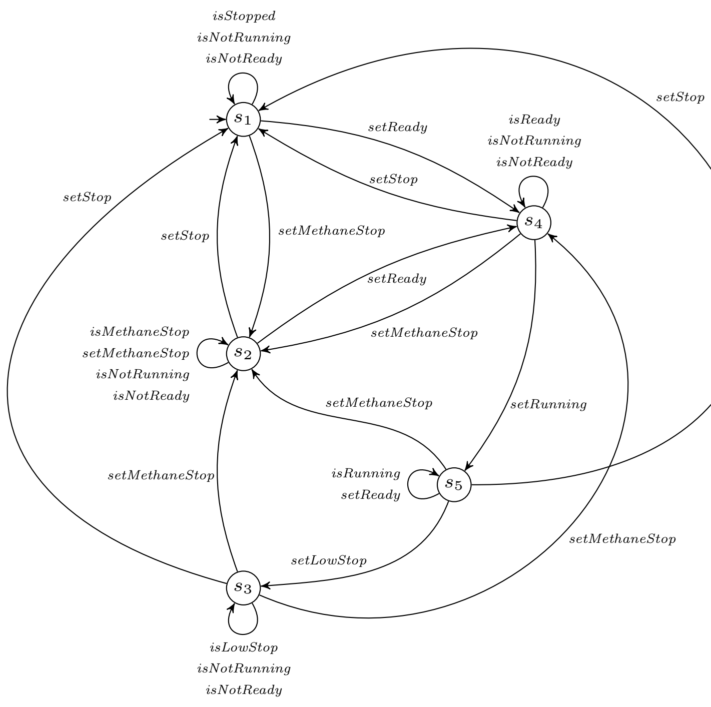


Fig. 12 The *state* FTS of the mine pump model from [36]

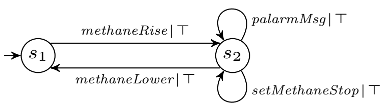


Fig. 13 FTS of the methane level environment from [36]

Listing 6 reports the result of applying our static analysis algorithm to the *system* FTS. The FTS contains no dead transitions, but 25 false optional transitions, among which $(s_7, \mathit{levelMsg}, l, s_{20})$, and one hidden deadlock state, viz. s_{20}. Indeed, state s_{20} is reachable in all products upon the execution of two must transitions (the second one being the false optional transition $(s_7, \mathit{levelMsg}, l, s_{20})$), while s_{20} is a deadlock in all 8 products that lack any of the features from the subset $\{ll, ln, lh\}$.

Listing 6 Result of the static analysis on the FTS of Figure 11

```
MINE PUMP: not live
 LIVE STATES = [S6,S7,...,S19,S21,S22,...,S30]
 DEAD TRANSITIONS = []
 FALSE OPTIONAL TRANSITIONS = [(S7,S20,levelMsg),(S9,S10,isRunning),...,
  (S29,S30,setLowStop)]
 HIDDEN DEADLOCK STATES = [S20]
```

Hence the *system* FTS is ambiguous, but it would suffice to turn its false optional transitions into must transitions and to add an explicit deadlock state $s_\dagger$ and a transition $(s_{20}, \dagger, \neg ll \wedge \neg ln \wedge \neg lh, s_\dagger)$ to make the *system* FTS unambiguous. Actually, a deadlock often indicates an error in the modelling, either in the feature model or in the behavioural model, i.e., the FTS. In fact, another solution to make the *system* FTS unambiguous would be to slightly change the feature model, e.g., by requiring the presence of at least one of the features ll, ln, or lh via an or-relationship. Doing so, the feature model becomes $\phi = (c \leftrightarrow (ct \vee cp)) \wedge l \wedge (ll \vee ln \vee lh)$, thus resulting in 56 products (i.e., excluding the 8 products over F that satisfy $(c \leftrightarrow (ct \vee cp)) \wedge l$, but lack any of the features from the subset $\{ll, ln, lh\}$). In [35,40,36], instead, an alternative feature model in which only c (and implicitly ct and cp) and m are optional was considered, resulting in only the four products over F that satisfy $(c \leftrightarrow (ct \wedge cp)) \wedge l \wedge ll \wedge ln \wedge lh$.

Yet another solution to make the *system* FTS unambiguous would be to slightly change the FTS itself, to make sure that it contains neither a hidden nor an explicit deadlock state. In this case, it would suffice to add one or more transitions to leave state s_{20} in a meaningful way. This is the solution opted for in [39,61,60,24,62], which use the specification in fPromela of the complete mine pump model as originally distributed with SNIP [37] and its re-engineered successor ProVeLines [42] (https://bitbucket.org/maxcordy/provelines-cora/) or their translations for mCRL2 [46,31] (http://www.mcrl2.org/) or VMC [20,19] (http://fmt.isti.cnr.it/vmc/). Basically, three transitions are added to the *system* FTS of Figure 11 from state s_{20} to the initial state s_6 to cover the cases in which features from the subset $\{ll, ln, lh\}$ are missing, viz. $(s_{20}, \mathit{highLevel}, \neg lh, s_6)$, $(s_{20}, \mathit{lowLevel}, \neg ll, s_6)$, and $(s_{20}, \mathit{normalLevel}, \neg ln, s_6)$.

The false optional transitions and the hidden deadlock state of the *system* FTS are propagated into the controller of the mine pump model, which we recall to be the parallel composition of the *system* and *state* FTSs. Application of our static analysis algorithm to the FTS of the controller of the mine pump model reports that the FTS contains no dead transitions, 59 false optional transitions, and 4 hidden deadlock states. The situation is different for the complete mine pump model, which we recall to be the parallel composition of five FTSs, viz. the *system* and *state* FTSs and three further FTSs that model a water pump and water and methane level sensors. From the FTS of the methane level, depicted in Figure 13, we immediately note that the actions *methaneRise* and *methaneLower* are local actions of this FTS that do not synchronise with any of the other four FTSs. Hence, while the solutions suggested above would make the *system* FTS of Figure 11 unambiguous, it is clear that the FTS of the complete mine pump model is deadlock-free, since it can indefinitely execute the sequence of actions *methaneRise* followed by *methaneLower*. This is confirmed by our static analysis algorithm applied to the FTS of the complete mine pump model, which reports that the FTS contains no dead transitions and no hidden deadlock states, but a stunning 308 false optional transitions.[8] The fact that the *system* FTS has hidden deadlock states that are no longer present in the FTS of the complete mine pump model demonstrates the usefulness of analysing component FTSs in isolation.

In general, while the parallel composition of unambiguous FTSs does not introduce false optional transitions, the composite FTS may contain dead transitions or hidden deadlock states. We have seen an example of the introduction of dead transitions in the composite FTS of the coffee and soup component, whose individual FTSs did not exhibit dead transitions.

The application of the static analysis algorithm to individual component FTSs is surely desirable as it results in less ambiguous specifications of the components constituting a composed system, and it possibly allows more efficient model checking of the composed system (more on this in the next section, cf. the part of Fig. 4 that is not red). A further advantage is that our approach becomes applicable also to feature-oriented systems composed by superimposition, since in [63] it is shown how to transform feature-oriented systems composed by parallel composition into feature-oriented systems composed by superimposition while maintaining behaviour and modularity.

Instead, the application of the static analysis algorithm to a composed FTS resulting from the parallel composition of several FTSs is less desirable because the benefits of detecting ambiguities are greatly reduced. This is due to the lack of a detailed specification of the composed FTS, which is merely a semantic model without a matching syntactic specification. Note that composed configurable systems can also be described as Multi SPLs (MPLs), i.e., sets of interdependent SPLs [71]. It is not clear how to obtain results for composed FTSs by reusing results of analyses performed in isolation on its component FTSs, in analogy with recently proposed compositional approaches for analysing MPLs [80,47].

Claroline We conclude this section with a very large (monolithic) system. The Claroline SPL is a configurable system whose FTS model, originally introduced in [50], was reverse-engineered from an Apache weblog (containing 12,689,030 HTTP requests) of a dynamically configurable course platform used at the University of Namur. The FTS model has since been used in several publications [56,52,53,54,49,55]. The Claroline SPL has 44 features and its feature model is quite large: it is represented by a formula with 299 logical connectives (omitted here), resulting in more than 5,000,000 product configurations. The FTS of Claroline contains 107 states and 11,236 transitions. Application of our static analysis algorithm reports (after running for about one hour) that the FTS contains no dead transitions and no hidden deadlock states, but 259 false optional transitions. Note that, since the FTS of Claroline has been generated from the analysis of actual execution paths, the discovery of dead transitions would have immediately signalled some major bug either in the feature model or in the feature expressions, or in the log analysis procedure.

[8] The FTS of the complete mine pump model could not be analysed in a reasonable amount of time with the static analysis algorithm presented in [9].

Table 1 Characteristics of the FTSs considered in this paper and results of static analysis

FTS	characteristics			results of static analysis				computational effort	
Model	$\lvert S\rvert$	$\lvert\delta\rvert$	$\lvert\Sigma\rvert$	live-ness	# dead transitions	# false optional transitions	# hidden deadlock states	run-time (s)	memory usage (Mb)
Vending machine [36]	9	13	12	yes	0	6	0	0.26	29.765
Coffee machine [5]	14	23	15	yes	0	14	0	0.29	30.305
Soup component [25]	13	28	18	yes	0	7	0	0.316	30.85
Mine pump (system) [36]	25	41	22	no	0	25	1	0.344	31.704
Mine pump (controller) [36]	77	104	22	no	0	59	4	0.548	36.295
Coffee/Soup machine [25]	182	691	33	yes	8	284	0	37.766	119.427
Mine pump (complete) [36]	418	1,255	26	yes	0	308	0	98.994	119.127
Claroline [50]	107	11,236	106	yes	0	259	0	2413.8	2010.229

6.2 Performance results

In Table 1, we report some data concerning the static analyses of the FTSs discussed above.

The FTSs of the vending machine, the coffee machine, and the soup component are all live (i.e., no deadlocks), with no dead transitions, while a respective 46%, 61%, and 25% of their transitions are false optional. Their static analyses are immediate. Also the FTS of the coffee/soup machine is live, but 41% and 1% of its transitions are false optional and dead, respectively. Its static analysis takes about a minute. The static analysis of the *system* FTS of the mine pump and that of the mine pump controller (i.e., the parallel composition of the *system* FTS and the *state* FTS) are immediate, but neither of these FTSs is live because 4% and 5% of their states, respectively, are hidden deadlocks. None of their transitions are dead, but 61% and 57% are false optional, respectively. Instead, the FTS of the complete mine pump is live and it has no dead transitions, but 25% of its transitions are false optional. Its analysis is not immediate, but takes a few minutes. Recall that this analysis could not be performed in a reasonable amount of time with the static analysis algorithm from [9]. The FTS of Claroline, finally, requires about an hour to analyse. It is live and it has no dead transitions, but 2% of its transitions are false optional.

Next, we compare the current implementation of the static analysis algorithm, as introduced in Listing 1, with the implementation used in [9], where an algorithm looks for all simple paths from the initial state to each state by visiting all cycle-free paths (starting from the initial state) in a depth-first manner. The results are reported in Table 2, where timeout stands for 'aborted after more than 2 hours'. The results show a clear improvement in runtime, ranging from a 3.54x speedup for the FTS of the vending machine to speedups of $>$ 7200x for the three largest FTSs. This demonstrates the improved efficiency of the current implementation.

Table 2 Comparison of current implementation of static analysis algorithm with that in [9]

FTS	characteristics			computational effort				results
				implementation in [9]		current implementation		
Model	|S|	|δ|	|Σ|	runtime (s)	memory usage (Mb)	runtime (s)	memory usage (Mb)	runtime speedup
Vending machine [36]	9	13	12	0.92	38.230	0.26	29.765	3.54x
Coffee machine [5]	14	23	15	2.822	40.140	0.29	30.305	9.72x
Soup component [25]	13	28	18	2.544	40.870	0.316	30.85	8.05x
Mine pump (system) [36]	25	41	22	2.192	41.899	0.344	31.704	6.37x
Mine pump (controller) [36]	77	104	22	8.12	49.091	0.548	36.295	14.82x
Coffee/Soup machine [25]	182	691	33	timeout	–	37.766	119.427	>7200.00x
Mine pump (complete) [36]	418	1,255	26	timeout	–	98.994	119.127	>7200.00x
Claroline [50]	107	11,236	106	timeout	–	2413.8	2010.229	>7200.00x

Table 3 Comparison of static analysis implementations for liveness of the largest FTSs

FTS	characteristics			computational effort				results
				current implementation		specialised implementation		
Model	|S|	|δ|	|Σ|	runtime (s)	memory usage (Mb)	runtime (s)	memory usage (Mb)	runtime fraction
Coffee/Soup machine [25]	182	691	33	37.766	119.427	2.288	61.620	6.06%
Mine pump (complete) [36]	417	1,255	26	98.994	119.127	2.948	68.969	2.97%
Claroline [50]	107	11,236	106	2413.8	2010.229	86.752	551.888	3.59%

In Table 3, we report a comparison of the current implementation of the static analysis algorithm with a specialised implementation that only detects hidden deadlocks, applied to the three largest FTSs showcased in this section. This specialised implementation refers to the first part (lines 1–10) of the static analysis algorithm in Listing 1, which (as pointed out at the end of Section 5) represents a hidden deadlocks discovery algorithm (i.e., analysing only liveness). The results show that only a fraction of the runtime of the current implementation is needed for deadlock detection, ranging from 6.06% for the FTS of the coffee/soup machine to only 2.97% and 3.59% for the complete mine pump and Claroline FTSs, respectively.

All the experiments presented in this section were performed on a Mac Pro (Late 2013) 3.7 Ghz Quad-Core with an Intel Xeon E5 processor with 10 Mb L3 cache and 64 Gb (four 16 Gb) of 1866 Mhz DDR3 ECC memory. All the experiments were performed five times each and the average time and memory usage of each was collected and reported in the tables. We used Python 3.6.

7 Family-based Verification

In analogy with anomaly detection in feature models, dead featured transitions in an FTS clearly indicate a modelling error, whereas false optional featured transitions often provide a wrong idea of the domain by giving the impression that certain behaviour is optional while actually it is mandatory (i.e., it occurs in all products of the FTS). However, our engineering methodology (i.e., the top-right red part of Fig. 4) that allows the transformation of an ambiguous FTS into an unambiguous FTS also serves another purpose, viz. to facilitate a kind of family-based model checking of properties expressed as logic formulas. As anticipated in earlier sections, according to the strategy outlined in Figure 4 (the part that is not red), a property ϕ specified in either LTL or v-ACTLive$^\Box$ can be verified (with a linear complexity) directly on an unambiguous FTS $\mathcal{F}$ (ignoring its feature expressions) such that ϕ holds for all product LTSs in $\mathsf{lts}(\mathcal{F})$ whenever it holds for $\mathcal{F}$. This strategy offering a number of efficient verification options is another contribution of this paper.

An FTS that has no hidden deadlocks is said to be *live*. In this section, we show that a live FTS enjoys the property that all valid linear-time LTL formulas are preserved by all its products, as well as all valid branching-time v-ACTLive$^\Box$ properties (as we already showed in [9]). Intuitively, these results are based on the fact that all transitions (and thus paths) in products of an FTS $\mathcal{F}$, i.e. LTSs in $\mathsf{lts}(\mathcal{F})$, also occur in $\mathcal{F}$.

Branching-time Properties To start with the latter, v-ACTLive$^\Box$ is a rich fragment of the variability-aware action-based and state-based branching-time modal temporal logic v-ACTL and it is interpreted on so-called 'live' MTSs [12, 7,13,14]. A Modal Transition System (MTS) is an LTS that distinguishes admissible ('may'), necessary ('must'), and optional (may but not must) transitions such that by definition all necessary and optional transitions are also admissible [77,75]. In [13], an MTS is defined to be *live* if all its states are live, where a live state of an MTS is such that it does not occur as a final state in any of its products (these are LTSs obtained from the MTS in a way similar to Definition 4), resulting in an MTS in which every path is infinite. Then it is proved that the validity of formulas expressed in v-ACTLive$^\Box$ is preserved in all products (cf. [13, Theorem 4]), thus allowing a kind of family-based model checking of MTSs. It is not difficult to see that this result continues to hold for MTSs whose every state is either live or final.

Note that any FTS $\mathcal{F}$ can be transformed into an MTS $\mathcal{F}_{\mathrm{MTS}}$ by considering its must transitions as necessary transitions, its featured transitions as optional transitions, and all its transitions as admissible, and by removing all feature expressions. If $\mathcal{F}$ is live, then $\mathcal{F}_{\mathrm{MTS}}$ is live, with respect to the FTS's set of products $\mathsf{lts}(\mathcal{F})$, because it has no hidden deadlocks.[9]

[9] From VMC v6.4 onwards, final states of an MTS are no longer considered 'hidden deadlocks' (i.e., they are considered live states) since they are deadlocks but not at all 'hidden'. For such MTSs, the same preservation properties of [13] apply.

Moreover, all transitions of $\mathcal{F}$ whose corresponding (LTS) transitions are mandatorily present in all products correspond to necessary transitions in $\mathcal{F}_{\text{MTS}}$. This demonstrates that the above mentioned result from [13] can be carried over to live FTSs, thus allowing a kind of family-based model checking of such FTSs for the v-ACTL fragment v-ACTLive$^{\Box}$. Hence, the following result holds, where $\models$ denotes the satisfaction relation of v-ACTLive$^{\Box}$ interpreted over MTSs.

Proposition 1 ([9]) *Any formula ϕ of v-ACTLive$^{\Box}$ is preserved by live FTSs: given a live FTS $\mathcal{F}$, whenever $\mathcal{F}_{MTS} \models \phi$, then $\mathcal{F}|_{\lambda} \models \phi$ for all products $\mathcal{F}|_{\lambda} \in \mathsf{lts}(\mathcal{F})$.*

Furthermore, note that states in an FTS that are the source of at least one must transition are by definition live. Hence, replacing all redundant feature expressions of false optional transitions (syntactic sugar) with $\top$ results in more must transitions, thus allowing for a more efficient kind of (family-based) verification.

Linear-time Properties In addition to [9], here we also consider linear-time properties. As said before, a live FTS also enjoys the property that all valid linear-time LTL formulas are preserved by all its products. This can be seen as follows. A path in an LTS is said to be *maximal* if it cannot be extended further, i.e. it is infinite or it ends in a deadlock state. Model checking LTL formulas on an LTS reduces to analysing its maximal paths: an LTL formula is valid if it holds for all maximal paths. These notions trivially carry over to FTSs by ignoring their feature expressions. Clearly, if an FTS is live, i.e. it has no hidden deadlocks, then the set of maximal paths of any product (LTS) is a subset of the set of maximal paths of the FTS. Hence, the following result holds, where $\mathcal{F}_{\text{LTS}}$ denotes the LTS obtained from an FTS $\mathcal{F}$ by removing its feature expressions and $\models$ denotes the satisfaction relation of LTL interpreted over LTSs.

Proposition 2 *Any formula ϕ of LTL is preserved by live FTSs: given a live FTS $\mathcal{F}$, whenever $\mathcal{F}_{LTS} \models \phi$, then $\mathcal{F}|_{\lambda} \models \phi$ for all products $\mathcal{F}|_{\lambda} \in \mathsf{lts}(\mathcal{F})$.*

The results presented in Propositions 1 and 2 show specific cases in which verification of live FTSs reduces to verification (with a linear complexity) of corresponding MTSs and LTSs that are obtained straightforwardly by ignoring the feature expressions (and distinguishing necessary and optional transitions in MTSs). This is made possible by the engineering methodology sketched in Figure 4 (the top-right red part). However, as illustrated by the strategy outlined in Figure 4 (the part that is not red), if either the property under verification is not an LTL or v-ACTLive$^{\Box}$ formula or the result of the verification is false, then the formula needs to be verified with classical (family-based) model checking.

In the remainder of this section, we apply these results to example FTSs from Section 6 and provide examples of v-ACTLive$^{\Box}$ and LTL formulas to

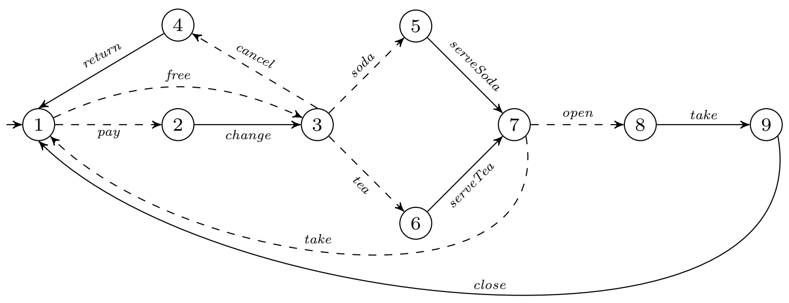


Fig. 14 MTS obtained from the FTS of Figure 8

illustrate their impact. The models allowing the verification of the example properties presented in this section are publicly available [10].

It is worth noticing that if we are interested in just the liveness of FTSs (e.g. to enable family-based model checking of invariant properties), then the first part (lines 1–10) of the static analysis algorithm in Listing 1 allows to establish the liveness of FTSs in a much more efficient way (cf. Table 3). Recall that this part represents a hidden deadlocks discovery algorithm, which we referred to as the specialised implementation in Section 6.

Vending Machine We have seen in Section 4.2 how to transform an ambiguous FTS into an unambiguous one. Furthermore, we have seen above how to transform an FTS into an MTS. In Figure 14, we depict the MTS [10] that is obtained in this way from the unambiguous (and thus live) FTS (described in the beginning of Section 6) that corresponds to the FTS of Figure 8.

As we argued in the beginning of this section, the resulting MTS is live, with respect to the FTS's set of products, thus allowing family-based model checking for v-ACTLive$^\square$ (cf. Proposition 1). In fact, v-ACTLive$^\square$ formulas can efficiently be verified on MTSs with the variability model checker VMC (http://fmt.isti.cnr.it/vmc), which is a tool for the analysis of branching-time properties over behavioural SPL models specified as an MTS with a set of logical variability constraints (akin to feature expressions) [20,19].

Originally, VMC used the variability constraints associated with the MTS to dynamically evaluate the liveness of each node. Based on [9], where we showed how to establish a priori the liveness of all nodes of an FTS, and thus of the MTS that can be obtained by transformation, the most recent prototypical extension of VMC, version 6.5, offers users the possibility to state explicitly that an MTS is live.

The input language of VMC is a process algebra. Listing 7 contains the specification of the vending machine in the process-algebraic input language accepted by VMC. Note that the system part or process model (i.e. without the constraints) can be seen as the natural encoding of the graph (MTS) of Figure 14, with the process terms corresponding to the states of the graph

[10] Dashed edges depict optional transitions and solid edges depict necessary transitions.

and `SYS` indicating the initial state. Intuitively, `a.P` models a process that executes action `a` and then behaves as `P`, while `P + Q` models a process that non-deterministically chooses to behave as either `P` or `Q`. Information on the modality of the transitions (`may`, `must`) is defined as a special additional parameter associated to the basic actions of the algebra, the default being must. Finally, `Constraints { LIVE }` explicitly declares that the MTS is live, the novel feature of VMC v6.5.

Listing 7 Specification in VMC of MTS of Figure 8

```
C1 = pay(may).C2 + free(may).C3
C2 = change.C3
C3 = cancel(may).C4 + soda(may).C5 + tea(may).C6
C4 = return.C1
C5 = serveSoda.C7
C6 = serveTea.C7
C7 = take(may).C1 + open(may).C8
C8 = take.C9
C9 = close.C1

SYS = C1

Constraints { LIVE }
```

Example formulas of branching-time properties of the vending machine that we verified in a kind of family-based manner with VMC include the following:

1. $AG\ AF_{pay \vee free}\ \top$: *infinitely often, either action pay or action free is executed;*
2. $AG\ [open]\ AF_{close}\ \top$: *it is always the case that the execution of action open is eventually followed by that of action close;*
3. $AG\ AF_{cancel \vee serveSoda \vee serveTea}\ \top$: *infinitely often, either action cancel or action serveSoda or action serveTea is executed;*
4. $\neg E\ [\top\ {}_{\neg tea}U_{serveTea}\ \top]$: *it is not possible that action serveTea is executed without being preceded by an execution of action tea;*
5. $[pay]\ AF_{take \vee cancel}\ \top$: *whenever action pay is executed, eventually also either action take or action cancel is executed.*[11]

Obviously, there are also numerous formulas of linear-time properties of the vending machine that can be verified in such kind of family-based manner, with tools such as SPIN (http://spinroot.com/). Example formulas include the following:[12]

1. $\square\,(selected \Rightarrow \diamond served)$: *after selecting a beverage, the machine will always eventually serve a beverage;*
2. $\square\,(served \Rightarrow \diamond taken)$: *after a beverage is served, the customer will always eventually take the beverage.*

[11] Abusing notation, this concerns execution of transition $(8, take, 9)$, not of $(7, take, 1)$.

[12] In [40,39], the states of an FTS are labelled with atomic propositions, omitted in figures to avoid clutter. For LTL model checking with SPIN, we assume that states 5 and 6 of the FTS depicted in Figure 8 are labelled with the proposition *selected*, state 7 with the proposition *served*, and states 1 and 9 with the proposition *taken*.

VMC v6.5 has thus been tailored for family-based model checking of temporal logic properties on FTSs (via their transformation in MTSs). At present, efficient SPL model checking on FTSs can be achieved by using dedicated family-based model checkers such as the ProVeLines [42] tool suite (including its predecessor SNIP [37]) or fNuSMV [38], or, alternatively, by using one of the highly optimised off-the-shelf model checkers like SPIN or mCRL2, which have recently been made amenable to family-based SPL model checking on FTSs [61,24,60,62,18].

Mine Pump We have seen in Section 6 that the complete mine pump model (an FTS with 418 states and 1,255 transitions) is live, thus allowing a kind of family-based model checking for v-ACTLive$^\Box$ and LTL (cf. Propositions 1 and 2). In fact, we have done so for the complete mine pump model specification in fPromela, as distributed with SNIP and ProVeLines, and its translation for VMC (recall that the model specifications are publicly available [10]).

Example formulas of branching-time properties of the complete mine pump that we verified in such kind of family-based manner with VMC include the following:[13]

1. $\textsc{max}\ X : (EX^{\Box}_{methaneRise \vee methaneLower}\ X)$: *the system behaviour includes a (mandatory) path that contains only variations of the methane level;*
2. $AG\,[palarmMsg]\ \neg E\ [\top\ _{\neg setMethaneStop}U_{palarmMsg}\ \top]$: *it is not possible that two palarmMsg actions occur without a setMethaneStop in between;*
3. $AG\ [highLevel]\ \neg E\ [\top\ _{\neg pumpStart}U_{lowLevel}\ \top]$: *it is not possible that the water level decreases if the pump did not start;*
4. $AG\ \neg(pumpoff \wedge EX_{pumpStop}\ \top)$: *a pumpStop action cannot occur if the pump is already off;*
5. $AG\ \neg(\neg ready \wedge EX_{pumpStart}\ \top)$: *a pumpStart action cannot occur if the system is not ready;*
6. $AG\ [stopCmd]\ \neg E\ [\neg stopped\ U_{startCmd}\ \top]$: *a start command cannot follow a stop command if in the meantime the system did not stop.*

Example formulas of linear-time properties of the complete mine pump model that we verified in such kind of family-based manner with SPIN include the following:[14]

1. $\Box\ (\neg pumpOn \vee stateRunning)$: *if the pump is on, the actual pump state is set to running;*

[13] In [35,36], the states of the FTSs constituting the complete mine pump model are labelled with atomic propositions. In particular, the initial state of the FTS modelling the water pump, not depicted here, is labelled with proposition *pumpoff*, while states s_1 and s_4 of the FTS depicted in Figure 12 are labelled with propositions *ready* and *stopped*, respectively.

[14] In the fPromela specification of the complete mine pump model distributed with SNIP and ProVeLines, *pumpOn* and *methane* are Booleans that are set to true when the pump is turned on or methane is detected, respectively, whereas the remaining variables are macros (e.g. *stateRunning* defines that the FTS depicted in Figure 12 is in state s_5, *readCommand* defines that the FTS depicted in Figure 11 has received a *commandMsg*, and *highWater* defines that the FTS has received a *levelMsg* stating that the water level is high).

2. $((\Box\Diamond\, \mathit{readCommand}) \wedge (\Box\Diamond\, \mathit{readAlarm}) \wedge (\Box\Diamond\, \mathit{readLevel})) \Rightarrow (\neg\Diamond\Box\,(\neg \mathit{pumpOn} \wedge \neg \mathit{methane} \wedge \mathit{highWater}))$: *if the controller can fairly receive each of the three message types, then the pump is never indefinitely off when the water is high*;
3. $\Box\,((\neg \mathit{pumpOn} \wedge \mathit{lowWater} \wedge \Diamond \mathit{highWater}) \Rightarrow ((\neg \mathit{pumpOn})\ U\ \mathit{highWater}))$: *when the pump is off and the water is low, it will only start once the water is high again.*

These are precisely the properties #18, #34, and #41, respectively, as verified with both SNIP and SPIN in [39].

Toolchain In [11], we present FTS4VMC, a tool developed specifically as a front-end for VMC with a user-friendly GUI. The resulting toolchain allows a modeller to analyse an FTS for ambiguities, remove them, transform the resulting live FTS into an MTS and perform an efficient kind of family-based model checking of v-ACTLive$^\Box$ properties. The FTS4VMC implementation is publicly available from https://github.com/fts4vmc/FTS4VMC.

8 Conclusion

In this paper, we have revisited several types of static analysis that can be performed over an FTS as part of an engineering methodology. Concretely, we analyse FTSs for hidden deadlocks and anomalies in the form of false optional and dead transitions. The removal of hidden deadlocks improves the clarity of FTSs and enables an efficient kind of family-based model checking of live FTSs. Dead transitions identify real modelling errors present in FTSs that should be removed. False optional transitions reveal redundancies in the feature expressions labelling the FTSs. Moreover, replacing such syntactic sugar by $\top$ leads to more must transitions, which eases verification and may increase the set of properties verifiable by v-ACTLive$^\Box$. We have presented a new algorithm for these static analyses of FTSs, for which we have proved the correctness. We have evaluated the suitability of the new algorithm by applying it to a large number of exemplary FTSs from the literature, and we have also showed the usefulness for an efficient kind of family-based model checking of FTSs. In particular, we have empirically demonstrated the superiority of the new algorithm with respect to the algorithm presented in [9], by making feasible (in reasonable time) the static analysis of FTSs of considerable size (cf. Table 2).

The python code implementing the algorithm and the specifications of all the models that are needed to reproduce the experiments presented in Sections 6 (static analysis) and 7 (verification) are publicly available [10]. Also the implementation of the FTS4VMC tool developed specifically as a front-end for VMC is publicly available (cf. Section 7). A front-end tool for SPIN, based on a transformation from FTSs to PROMELA, is ongoing work.

In principle, our static analysis checks could all be performed by classical family-based model-checking approaches by expressing the ambiguity properties in CTL (exploiting the fact that they concern reachability questions).

However, verifying such properties for each state and transition of an FTS requires a considerable number of verifications. Moreover, the complexity of verifying a single CTL formula on an FTS is exponential in the number of features [38]. Note that the kind of family-based model checking we make possible is linear in the size of the LTS or MTS that is obtained from a live FTS (by ignoring its feature expressions). Nevertheless, we intend to investigate this issue in more detail, also empirically, possibly exploiting symbolic representations.

Recently [81], a subset of the authors proposed an approach and a tool for checking SPLs of statecharts [69]. The tool checks that all the products can be generated and are well-formed statecharts. In future work, we would like to extend it by adding behavioural ambiguity detection analyses like the ones presented in this paper. We also would like to study how to adapt our static analysis algorithms to apply them to high-level SPL modelling languages (e.g. fPROMELA [37,39] and fNuSMV [38]). Finally, it would be interesting to mechanise our formalisations and associated proofs to provide further evidence of the soundness of our static analysis techniques.

Acknowledgements We thank the anonymous reviewers for useful comments and suggestions that helped us to improve the presentation.

References

1. Alpern, B., Schneider, F.B.: Defining liveness. Inf. Process. Lett. **4**(21), 181–185 (1985). https://doi.org/10.1016/0020-0190(85)90056-0
2. Apel, S., Batory, D.S., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines: Concepts and Implementation. Springer (2013). https://doi.org/10.1007/978-3-642-37521-7
3. Asirelli, P., ter Beek, M.H., Fantechi, A., Gnesi, S.: Deontic Logics for Modeling Behavioural Variability. In: D. Benavides, A. Metzger, U. Eisenecker (eds.) Proceedings of the 3rd International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'09), *ICB Research Report*, vol. 29, pp. 71–76. Universität Duisburg-Essen (2009)
4. Asirelli, P., ter Beek, M.H., Fantechi, A., Gnesi, S.: A Logical Framework to Deal with Variability. In: D. Méry, S. Merz (eds.) Proceedings of the 8th International Conference on Integrated Formal Methods (IFM'10), *LNCS*, vol. 6396, pp. 43–58. Springer (2010). https://doi.org/10.1007/978-3-642-16265-7_5
5. Asirelli, P., ter Beek, M.H., Fantechi, A., Gnesi, S.: Formal Description of Variability in Product Families. In: Proceedings of the 15th International Software Product Lines Conference (SPLC'11), pp. 130–139. IEEE (2011). https://doi.org/10.1109/SPLC.2011.34
6. Audemard, G., Lagniez, J.M., Szczepanski, N., Tabary, S.: An Adaptive Parallel SAT Solver. In: M. Rueher (ed.) Proceedings of the 22nd International Conference on Principles and Practice of Constraint Programming (CP'16), *LNCS*, vol. 9892, pp. 30–48. Springer (2016). https://doi.org/10.1007/978-3-319-44953-1_3
7. ter Beek, M.H., Damiani, F., Gnesi, S., Mazzanti, F., Paolini, L.: From Featured Transition Systems to Modal Transition Systems with Variability Constraints. In: R. Calinescu, B. Rumpe (eds.) Proceedings of the 13th International Conference on Software Engineering and Formal Methods (SEFM'15), *LNCS*, vol. 9276, pp. 344–359. Springer (2015). https://doi.org/10.1007/978-3-319-22969-0_24
8. ter Beek, M.H., Damiani, F., Gnesi, S., Mazzanti, F., Paolini, L.: On the expressiveness of modal transition systems with variability constraints. Sci. Comput. Program. **169**, 1–17 (2019). https://doi.org/10.1016/j.scico.2018.09.006

9. ter Beek, M.H., Damiani, F., Lienhardt, M., Mazzanti, F., Paolini, L.: Static Analysis of Featured Transition Systems. In: Proceedings of the 23rd International Systems and Software Product Line Conference (SPLC'19), pp. 39–51. ACM (2019). https://doi.org/10.1145/3336294.3336295
10. ter Beek, M.H., Damiani, F., Lienhardt, M., Mazzanti, F., Paolini, L.: Supplementary material for: "Static Analysis of Featured Transition Systems" (2019). https://doi.org/10.5281/zenodo.2616646
11. ter Beek, M.H., Damiani, F., Mazzanti, F., Scarso, G., Valfrè, M.: Static Analysis and Family-based Model Checking of Featured Transition Systems with VMC (2021). https://doi.org/10.5281/zenodo.4497888
12. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: Using FMC for Family-based Analysis of Software Product Lines. In: Proceedings of the 19th International Software Product Line Conference (SPLC'15), pp. 432–439. ACM (2015). https://doi.org/10.1145/2791060.2791118
13. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: Modelling and analysing variability in product families: Model checking of modal transition systems with variability constraints. J. Log. Algebr. Meth. Program. **85**(2), 287–315 (2016). https://doi.org/10.1016/j.jlamp.2015.11.006
14. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: States and Events in KandISTI: A Retrospective. In: T. Margaria, S. Graf, K.G. Larsen (eds.) Models, Mindsets, Meta: The What, the How, and the Why Not?, *LNCS*, vol. 11200, pp. 110–128. Springer (2019). https://doi.org/10.1007/978-3-030-22348-9_9
15. ter Beek, M.H., Legay, A., Lluch Lafuente, A., Vandin, A.: Quantitative Analysis of Probabilistic Models of Software Product Lines with Statistical Model Checking. Electron. Proc. Theor. Comput. Sci. **182**, 56–70 (2015). https://doi.org/10.4204/EPTCS.182.5
16. ter Beek, M.H., Legay, A., Lluch Lafuente, A., Vandin, A.: A framework for quantitative modeling and analysis of highly (re)configurable systems. IEEE Trans. Softw. Eng. **46**(3), 321–345 (2020). https://doi.org/10.1109/TSE.2018.2853726
17. ter Beek, M.H., Lluch Lafuente, A., Petrocchi, M.: Combining Declarative and Procedural Views in the Specification and Analysis of Product Families. In: Proceedings of the 17th International Software Product Line Conference (SPLC'13), vol. 2, pp. 10–17. ACM (2013). https://doi.org/10.1145/2499777.2500722
18. ter Beek, M.H., van Loo, S., de Vink, E.P., Willemse, T.A.: Family-Based SPL Model Checking Using Parity Games with Variability. In: H. Wehrheim, J. Cabot (eds.) Proceedings of the 23rd International Conference on Fundamental Approaches to Software Engineering (FASE'20), *LNCS*, vol. 12076, pp. 245–265. Springer (2020). https://doi.org/10.1007/978-3-030-45234-6_12
19. ter Beek, M.H., Mazzanti, F.: VMC: Recent Advances and Challenges Ahead. In: Proceedings of the 18th International Software Product Line Conference (SPLC'14), vol. 2, pp. 70–77. ACM (2014). https://doi.org/10.1145/2647908.2655969
20. ter Beek, M.H., Mazzanti, F., Sulova, A.: VMC: A Tool for Product Variability Analysis. In: D. Giannakopoulou, D. Méry (eds.) Proceedings of the 18th International Symposium on Formal Methods (FM'12), *LNCS*, vol. 7436, pp. 450–454. Springer (2012). https://doi.org/10.1007/978-3-642-32759-9_36
21. ter Beek, M.H., Reniers, M.A., de Vink, E.P.: Supervisory Controller Synthesis for Product Lines Using CIF 3. In: T. Margaria, B. Steffen (eds.) Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques (ISoLA'16), *LNCS*, vol. 9952, pp. 856–873. Springer (2016). https://doi.org/10.1007/978-3-319-47166-2_59
22. ter Beek, M.H., de Vink, E.P.: Towards Modular Verification of Software Product Lines with mCRL2. In: T. Margaria, B. Steffen (eds.) Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'14), *LNCS*, vol. 8802, pp. 368–385. Springer (2014). https://doi.org/10.1007/978-3-662-45234-9_26
23. ter Beek, M.H., de Vink, E.P.: Using mCRL2 for the Analysis of Software Product Lines. In: Proceedings of the 2nd FME Workshop on Formal Methods in Software Engineering (FormaliSE'14), pp. 31–37. IEEE (2014). https://doi.org/10.1145/2593489.2593493

24. ter Beek, M.H., de Vink, E.P., Willemse, T.A.C.: Family-Based Model Checking with mCRL2. In: M. Huisman, J. Rubin (eds.) Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering (FASE'17), *LNCS*, vol. 10202, pp. 387–405. Springer (2017). https://doi.org/10.1007/978-3-662-54494-5_23
25. Belder, T., ter Beek, M.H., de Vink, E.P.: Coherent branching feature bisimulation. Electron. Proc. Theor. Comput. Sci. **220**(3), 14–30 (2015). https://doi.org/10.4204/EPTCS.182.2
26. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated Analysis of Feature Models 20 Years Later: a Literature Review. Inf. Syst. **35**(6), 615–636 (2010). https://doi.org/10.1016/j.is.2010.01.001
27. Beohar, H., Varshosaz, M., Mousavi, M.R.: Basic behavioral models for software product lines: Expressiveness and testing pre-orders. Sci. Comput. Program. **123**, 42–60 (2016). https://doi.org/10.1016/j.scico.2015.06.005
28. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series. Springer (2004). https://doi.org/10.1007/978-3-662-07964-5
29. Bjørner, N., Phan, A.D., Fleckenstein, L.: νZ – An Optimizing SMT Solver. In: C. Baier, C. Tinelli (eds.) Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15), *LNCS*, vol. 9035, pp. 194–199. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_14
30. 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: Proceedings of the 34th Conference on Programming Language Design and Implementation (PLDI'13), pp. 355–364. ACM (2013). https://doi.org/10.1145/2491956.2491976
31. Bunte, O., Groote, J., Keiren, J., Laveaux, M., Neele, T., de Vink, E., Wesselink, W., Wijs, A., Willemse, T.: The mCRL2 Toolset for Analysing Concurrent Systems: Improvements in Expressivity and Usability. In: T. Vojnar, L. Zhang (eds.) Proc. TACAS'19, *LNCS*, vol. 11428, pp. 21–39. Springer (2019). https://doi.org/10.1007/978-3-030-17465-1_2
32. Castro, T.M., Lanna, A., Alves, V., Teixeira, L., Apel, S., Schobbens, P.: All roads lead to Rome: Commuting strategies for product-line reliability analysis. Sci. Comput. Program. **152**, 116–160 (2018). https://doi.org/10.1016/j.scico.2017.10.013
33. Chess, B., West, J.: Secure Programming with Static Analysis. Addison-Wesley (2007)
34. Chrszon, P., Dubslaff, C., Klüppelholz, S., Baier, C.: ProFeat: feature-oriented engineering for family-based probabilistic model checking. Form. Asp. Comp. **30**(1), 45–75 (2018). https://doi.org/10.1007/s00165-017-0432-4
35. Classen, A.: Modelling with fts: a collection of illustrative examples. Tech. Rep. P-CS-TR SPLMC-00000001, University of Namur (2010)
36. Classen, A.: Modelling and model checking variability-intensive systems. Ph.D. thesis, University of Namur (2011)
37. Classen, A., Cordy, M., Heymans, P., Legay, A., Schobbens, P.: Model checking software product lines with SNIP. Int. J. Softw. Tools Technol. Transf. **14**(5), 589–612 (2012). https://doi.org/10.1007/s10009-012-0234-1
38. Classen, A., Cordy, M., Heymans, P., Legay, A., Schobbens, P.Y.: Formal semantics, modular specification, and symbolic verification of product-line behaviour. Sci. Comput. Program. **80**(B), 416–439 (2014). https://doi.org/10.1016/j.scico.2013.09.019
39. Classen, A., Cordy, M., Schobbens, P., Heymans, P., Legay, A., Raskin, J.: Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. IEEE Trans. Softw. Eng. **39**(8), 1069–1089 (2013). https://doi.org/10.1109/TSE.2012.86
40. Classen, A., Heymans, P., Schobbens, P., Legay, A., Raskin, J.: Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In: Proceedings of the 32nd International Conference on Software Engineering (ICSE'10), pp. 335–344. ACM (2010). https://doi.org/10.1145/1806799.1806850
41. Cook, S.A.: The Complexity of Theorem-Proving Procedures. In: Proceedings of the 3rd Annual Symposium on Theory of Computing (STOC'71), pp. 151–158. ACM (1971). https://doi.org/10.1145/800157.805047

42. Cordy, M., Classen, A., Heymans, P., Schobbens, P.Y., Legay, A.: ProVeLines: A Product Line of Verifiers for Software Product Lines. In: Proceedings of the 17th International Software Product Line Conference (SPLC'13), vol. 2, pp. 141–146. ACM (2013). https://doi.org/10.1145/2499777.2499781
43. Cordy, M., Classen, A., Perrouin, G., Schobbens, P., Heymans, P., Legay, A.: Simulation-Based Abstractions for Software Product-Line Model Checking. In: Proceedings of the 34th International Conference on Software Engineering (ICSE'12), pp. 672–682. IEEE (2012). https://doi.org/10.1109/ICSE.2012.6227150
44. Cordy, M., Devroey, X., Legay, A., Perrouin, G., Classen, A., Heymans, P., Schobbens, P., Raskin, J.: A Decade of Featured Transition Systems. In: M.H. ter Beek, A. Fantechi, L. Semini (eds.) From Software Engineering to Formal Methods and Tools, and Back, *LNCS*, vol. 11865, pp. 285–312. Springer (2019). https://doi.org/10.1007/978-3-030-30985-5_18
45. Cordy, M., Schobbens, P., Heymans, P., Legay, A.: Beyond Boolean Product-Line Model Checking: Dealing with Feature Attributes and Multi-features. In: Proceedings of the 35th International Conference on Software Engineering (ICSE'13), pp. 472–481. IEEE (2013). https://doi.org/10.1109/ICSE.2013.6606593
46. Cranen, S., Groote, J.F., Keiren, J.J.A., Stappers, F.P.M., de Vink, E.P., Wesselink, W., Willemse, T.A.C.: An Overview of the mCRL2 Toolset and Its Recent Advances. In: N. Piterman, S.A. Smolka (eds.) Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13), *LNCS*, vol. 7795, pp. 199–213. Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_15
47. Damiani, F., Lienhardt, M., Paolini, L.: A formal model for Multi Software Product Lines. Sci. Comput. Program. **172**, 203–231 (2019). https://doi.org/10.1016/j.scico.2018.11.005
48. Delaware, B., Cook, W.R., Batory, D.: Fitting the pieces together: a machine-checked model of safe composition. In: Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'09), pp. 243–252. ACM (2009). https://doi.org/10.1145/1595696.1595733
49. Devroey, X., Perrouin, G., Cordy, M., Samih, H., Legay, A., Schobbens, P., Heymans, P.: Statistical prioritization for software product line testing: an experience report. Softw. Syst. Model. **16**(1), 153–171 (2017). https://doi.org/10.1007/s10270-015-0479-8
50. Devroey, X., Perrouin, G., Cordy, M., Schobbens, P., Legay, A., Heymans, P.: Towards Statistical Prioritization for Software Product Lines Testing. In: Proceedings of the 8th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'14), pp. 10:1–10:7. ACM (2014). https://doi.org/10.1145/2556624.2556635
51. Devroey, X., Perrouin, G., Legay, A., Cordy, M., Schobbens, P., Heymans, P.: Coverage Criteria for Behavioural Testing of Software Product Lines. In: T. Margaria, B. Steffen (eds.) Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'14), *LNCS*, vol. 8802, pp. 336–350. Springer (2014). https://doi.org/10.1007/978-3-662-45234-9_24
52. Devroey, X., Perrouin, G., Legay, A., Schobbens, P., Heymans, P.: Covering SPL Behaviour with Sampled Configurations: An Initial Assessment. In: Proceedings of the 9th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'15), pp. 59:59–59:66. ACM (2015). https://doi.org/10.1145/2701319.2701325
53. Devroey, X., Perrouin, G., Legay, A., Schobbens, P., Heymans, P.: Search-based Similarity-driven Behavioural SPL Testing. In: Proceedings of the 10th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'16), pp. 89–96. ACM (2016). https://doi.org/10.1145/2866614.2866627
54. Devroey, X., Perrouin, G., Papadakis, M., Legay, A., Schobbens, P., Heymans, P.: Featured Model-based Mutation Analysis. In: Proceedings of the 38th International Conference on Software Engineering (ICSE'16), pp. 655–666. ACM (2016). https://doi.org/10.1145/2884781.2884821
55. Devroey, X., Perrouin, G., Papadakis, M., Legay, A., Schobbens, P., Heymans, P.: Model-based mutant equivalence detection using automata language equivalence and simulations. J. Syst. Softw. **141**, 1–15 (2018). https://doi.org/10.1016/j.jss.2018.03.010

56. Devroey, X., Perrouin, G., Schobbens, P.: Abstract Test Case Generation for Behavioural Testing of Software Product Lines. In: Proceedings of the 18th International Software Product Line Conference (SPLC'14), vol. 2, pp. 86–93. ACM (2014). https://doi.org/10.1145/2647908.2655971
57. Dimovski, A.: CTL* family-based model checking using variability abstractions and modal transition systems. Int. J. Softw. Tools Technol. Transf. **22**(1), 35–55 (2020). https://doi.org/10.1007/s10009-019-00528-0
58. Dimovski, A., Legay, A., Wąsowski, A.: Variability Abstraction and Refinement for Game-Based Lifted Model Checking of Full CTL. In: R. Hähnle, W. van der Aalst (eds.) Proceedings of the 22nd International Conference on Fundamental Approaches to Software Engineering (FASE'19), *LNCS*, vol. 11424, pp. 192–209. Springer (2019). https://doi.org/10.1007/978-3-030-16722-6_11
59. Dimovski, A.S.: Abstract Family-Based Model Checking Using Modal Featured Transition Systems: Preservation of CTL*. In: A. Russo, A. Schürr (eds.) Proceedings of the 21st International Conference on Fundamental Approaches to Software Engineering (FASE'18), *LNCS*, vol. 10802, pp. 301–318. Springer (2018). https://doi.org/10.1007/978-3-319-89363-1_17
60. Dimovski, A.S., Al-Sibahi, A.S., Brabrand, C., Wąsowski, A.: Efficient family-based model checking via variability abstractions. Int. J. Softw. Tools Technol. Transf. **5**(19), 585–603 (2017). https://doi.org/10.1007/s10009-016-0425-2
61. Dimovski, A.S., Al-Sibahi, A.S., Brabrand, C., Wąsowski, A.: Family-Based Model Checking Without a Family-Based Model Checker. In: B. Fischer, J. Geldenhuys (eds.) Proceedings of the 22nd International SPIN Symposium on Model Checking of Software (SPIN'15), *LNCS*, vol. 9232, pp. 282–299. Springer (2015). https://doi.org/10.1007/978-3-319-23404-5_18
62. Dimovski, A.S., Wąsowski, A.: Variability-Specific Abstraction Refinement for Family-Based Model Checking. In: M. Huisman, J. Rubin (eds.) Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering (FASE'17), *LNCS*, vol. 10202, pp. 406–423. Springer (2017). https://doi.org/10.1007/978-3-662-54494-5_24
63. Dubslaff, C.: Compositional Feature-Oriented Systems. In: P.C. Ölveczky, G. Salaün (eds.) Proceedings of the 17th International Conference on Software Engineering and Formal Methods (SEFM'19), *LNCS*, vol. 11724, pp. 162–180. Springer (2019). https://doi.org/10.1007/978-3-030-30446-1_9
64. Dubslaff, C., Baier, C., Klüppelholz, S.: Probabilistic model checking for feature-oriented systems. In: S. Chiba, E. Tanter, E. Ernst, R. Hirschfeld (eds.) Transactions on Aspect-Oriented Software Development XII, *LNCS*, vol. 8989, pp. 180–220. Springer (2015). https://doi.org/10.1007/978-3-662-46734-3_5
65. Fantechi, A., Gnesi, S.: A behavioural model for product families. In: Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'07), pp. 521–524. ACM (2007). https://doi.org/10.1145/1287624.1287700
66. Fantechi, A., Gnesi, S.: Formal modeling for product families engineering. In: Proceedings of the 12th International Conference on Software Product Line Engineering (SPLC'08), pp. 193–202. IEEE (2008). https://doi.org/10.1109/SPLC.2008.45
67. Fischbein, D., Uchitel, S., Braberman, V.A.: A Foundation for Behavioural Conformance in Software Product Line Architectures. In: Proceedings of the ISSTA Workshop on Role of Software Architecture for Testing and Analysis (ROSATEA'06), pp. 39–48. ACM (2006). https://doi.org/10.1145/1147249.1147254
68. Gruler, A., Leucker, M., Scheidemann, K.D.: Modeling and Model Checking Software Product Lines. In: G. Barthe, F.S. de Boer (eds.) Proceedings of the 10th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'08), *LNCS*, vol. 5051, pp. 113–131. Springer (2008). https://doi.org/10.1007/978-3-540-68863-1_8
69. Harel, D.: Statecharts: a visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987). https://doi.org/10.1016/0167-6423(87)90035-9
70. Heule, M., Järvisalo, M., Suda, M.: The international sat competitions web page. https://www.satcompetition.org/. Accessed: 2019-03-22

71. Holl, G., Grünbacher, P., Rabiser, R.: A systematic review and an expert survey on capabilities supporting multi product lines. Inf. Softw. Technol. **54**(8), 828–852 (2012). https://doi.org/10.1016/j.infsof.2012.02.002
72. Hutter, F., Lindauer, M., Balint, A., Bayless, S., Hoos, H., Leyton-Brown, K.: The Configurable SAT Solver Challenge (CSSC). Artifi. Intell. **243**, 1–25 (2017). https://doi.org/10.1016/j.artint.2016.09.006
73. Kästner, C., Apel, S.: Type-checking Software Product Lines – A Formal Approach. In: Proceedings of the 23rd International Conference on Automated Software Engineering (ASE'08), pp. 258–267. IEEE (2008). https://doi.org/10.1109/ASE.2008.36
74. Kim, C.H.P., Batory, D.S., Khurshid, S.: Reducing Combinatorics in Testing Product Lines. In: Proceedings of the 10th International Conference on Aspect-Oriented Software Development (AOSD'11), pp. 57–68. ACM (2011). https://doi.org/10.1145/1960275.1960284
75. Křetínský, J.: 30 Years of Modal Transition Systems: Survey of Extensions and Analysis. In: L. Aceto, G. Bacci, G. Bacci, A. Ingólfsdóttir, A. Legay, R. Mardare (eds.) Models, Algorithms, Logics and Tools, *LNCS*, vol. 10460, pp. 36–74. Springer (2017). https://doi.org/10.1007/978-3-319-63121-9_3
76. Larsen, K.G., Nyman, U., Wąsowski, A.: Modal I/O Automata for Interface and Product Line Theories. In: R. De Nicola (ed.) Proceedings of the 16th European Symposium on Programming (ESOP'07), *LNCS*, vol. 4421, pp. 64–79. Springer (2007). https://doi.org/10.1007/978-3-540-71316-6_6
77. Larsen, K.G., Thomsen, B.: A Modal Process Logic. In: Proceedings of the 3rd Symposium on Logic in Computer Science (LICS'88), pp. 203–210. IEEE (1988). https://doi.org/10.1109/LICS.1988.5119
78. Lauenroth, K., Pohl, K., Töhning, S.: Model Checking of Domain Artifacts in Product Line Engineering. In: Proceedings of the 24th International Conference on Automated Software Engineering (ASE'09), pp. 269–280. IEEE (2009). https://doi.org/10.1109/ASE.2009.16
79. Liang, J.H., Ganesh, V., Czarnecki, K., Raman, V.: SAT-based Analysis of Large Real-world Feature Models is Easy. In: Proceedings of the 19th International Software Product Line Conference (SPLC'15), pp. 91–100. ACM (2015). https://doi.org/10.1145/2791060.2791070
80. Lienhardt, M., Damiani, F., Donetti, S., Paolini, L.: Multi Software Product Lines in the Wild. In: Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'18), pp. 89–96. ACM (2018). https://doi.org/10.1145/3168365.3170425
81. Lienhardt, M., Damiani, F., Testa, L., Turin, G.: On checking delta-oriented product lines of statecharts. Sci. Comput. Program. **166**, 3–34 (2018). https://doi.org/10.1016/j.scico.2018.05.007
82. Lochau, M., Mennicke, S., Baller, H., Ribbeck, L.: Incremental model checking of delta-oriented software product lines. J. Log. Algebr. Meth. Program. **85**(1), 245–267 (2016). https://doi.org/10.1016/j.jlamp.2015.09.004
83. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. Springer (1995). https://doi.org/10.1007/978-1-4612-4222-2
84. Mendonça, M., Wąsowski, A., Czarnecki, K.: SAT-based Analysis of Feature Models is Easy. In: Proceedings of the 13th International Software Product Line Conference (SPLC'09), pp. 231–240. ACM (2009)
85. de Moura, L.M., Bjørner, N.: Z3: An Efficient SMT Solver. In: C.R. Ramakrishnan, J. Rehof (eds.) Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08), *LNCS*, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
86. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer (2005). https://doi.org/10.1007/978-3-662-03811-6
87. Rhein, A.v., Liebig, J., Janker, A., Kästner, C., Apel, S.: Variability-Aware Static Analysis at Scale: An Empirical Study. ACM Trans. Softw. Eng. Methodol. **27**(4), 18:1–18:33 (2018). https://doi.org/10.1145/3280986
88. 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), 6:1–6:45 (2014). https://doi.org/10.1145/2580950

89. Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., Leich, T.: FeatureIDE: An extensible framework for feature-oriented software development. Sci. Comput. Program. **79**, 70–85 (2014). https://doi.org/10.1016/j.scico.2012.06.002
90. Vandin, A., ter Beek, M., Legay, A., Lluch Lafuente, A.: QFLan: A Tool for the Quantitative Analysis of Highly Reconfigurable Systems. In: K. Havelund, J. Peleska, B. Roscoe, E. de Vink (eds.) Proceedings of the 22nd International Symposium on Formal Methods (FM'18), *LNCS*, vol. 10951, pp. 329–337. Springer (2018). https://doi.org/10.1007/978-3-319-95582-7_19
91. Varshosaz, M., Beohar, H., Mousavi, M.R.: Basic behavioral models for software product lines: Revisited. Sci. Comput. Program. **168**, 171–185 (2018). https://doi.org/10.1016/j.scico.2018.09.001