Static analysis of featured transition systems

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

Static Analysis of Featured Transition Systems

Maurice H. ter Beek*
ISTI–CNR, Pisa, Italy
maurice.terbeek@isti.cnr.it

Ferruccio Damiani*
University of Turin, Turin, Italy
ferruccio.damiani@unito.it

Michael Lienhardt*
ONERA, Palaiseau, France
michael.lienhardt@onera.fr

Franco Mazzanti*
ISTI–CNR, Pisa, Italy
franco.mazzanti@isti.cnr.it

Luca Paolini*
University of Turin, Turin, Italy
luca.paolini@unito.it

ABSTRACT

A Featured Transition System (FTS) is a formal behavioural model for software product lines, which represents the behaviour of all the products of an SPL in a single compact structure by associating transitions with features that condition their existence in products. In general, an FTS may contain featured transitions that are unreachable in any product (so called *dead* transitions) or, on the contrary, mandatorily present in all products for which their source state is reachable (so called *false optional* transitions), as well as states from which only for certain products progress is possible (so called *hidden deadlocks*). In this paper, we provide algorithms to analyse an FTS for such ambiguities and to transform an ambiguous FTS into an unambiguous FTS. The scope of our approach is twofold. First and foremost, an ambiguous model is typically undesired as it gives an unclear idea of the SPL. Second, an unambiguous FTS paves the way for efficient family-based model checking. We apply our approach to illustrative examples from the literature.

CCS CONCEPTS

• **Software and its engineering → Specification languages**; **Formal methods**; **Software product lines**.

KEYWORDS

software product lines, formal specification, behavioural model, featured transition systems, static analysis

ACM Reference Format:
Maurice H. ter Beek, Ferruccio Damiani, Michael Lienhardt, Franco Mazzanti, and Luca Paolini. 2019. Static Analysis of Featured Transition Systems. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3336294.3336295

1 INTRODUCTION

Systems and Software Product Line Engineering (SPLE) advocates the reuse of components (systems as well as software) throughout all phases of product development. Following this paradigm, many businesses no longer develop single products, but rather a family or product line of closely-related, customisable products. This requires identifying the relevant features of the product domain to best exploit their commonality and manage their variability. A feature diagram or feature model defines those combinations of features that constitute valid product configurations [2].

While automated analysis of such structural variability models (e.g., detection of anomalies like dead or false optional features) has a 30-year history [20, 61], that of behavioural variability models has received considerable attention only during the last decade, following the seminal paper by Classen et al. [32]. Given that SPLs often concern massively (re)used and critical software (e.g., in smartphones and the automotive industry) it is important to demonstrate not only their correct configuration, but also their correct behaviour.

A Featured Transition System (FTS) is a formal behavioural model for SPLs, which represents the behaviour of all the products of an SPL in a single compact structure by associating transitions with features that condition their existence in products [30]. Quality assurance by means of model checking or testing is challenging, since ideally it must exploit the compact structure of the FTS to reason on the entire SPL at once. This is called family-based analysis, as opposed to enumerative product-based analysis in which every product is examined individually [60]. During the past decade, FTSs have shown to be amenable to family-based testing and model-checking [10, 11, 28–32, 34, 42–47, 52].

In this paper, we are interested in automated static analysis of FTSs. We want to catch (and offer a means to remove) possible ambiguities in FTSs, mimicking the anomaly detection as typically performed on feature models. In fact, an FTS may contain: *dead* transitions (i.e., featured transitions that are unreachable in any product); *false optional* transitions (i.e., featured transitions that are mandatorily present in all products for which their source state is reachable); and *hidden deadlocks* (i.e., states from which only for certain products progress is possible).

The contribution of this paper is a formalisation of ambiguities in FTSs and algorithms to analyse an FTS for such ambiguities and to transform an ambiguous FTS into an unambiguous one, as well as proofs of their correctness. The scope is twofold. First, in analogy with the anomalies in feature models, an ambiguous FTS is often undesired since it gives an unclear idea of the SPL. Second, an unambiguous FTS paves the way for efficient family-based model checking, because it has specific characteristics that enable model checking of properties expressed in a rich, action-based and variability-aware fragment of the well-known CTL logic directly on the FTS such that validity is preserved in all products.

*All authors contributed equally to this research.

SPLC '19, September 9–13, 2019, Paris, France

ACM ISBN 978-1-4503-7138-4/19/09…$15.00
https://doi.org/10.1145/3336294.3336295

We apply our approach to illustrative examples from the literature.

Related work. Static analysis of FTSs mimics the automated analysis of feature models [20, 61], by defining behavioural counterparts of dead and false optional features. It is related to static (program) analysis [24, 58], 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 [52], conventional static analysis techniques are applied to SPLs 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 [23], several well-known static analysis techniques are lifted to SPLs without the exponential blowup caused by generating and analysing all products individually, by converting such analyses to feature-sensitive analyses that operate on the entire SPL in one single pass. In [51], static type checking is extended from single programs to an entire SPL 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 for generating and compiling them first. An encompassing overview of (static) analysis strategies for SPLs can be found in [60] and a recent empirical study on applying variability-aware static analysis techniques to real-world configurable systems is presented in [59].

Outline. After some background (in Sect. 2), we define ambiguities in FTSs (in Sect. 3); provide criteria that enable to identify them by static analysis (in Sect. 4); present a static analysis algorithm to detect ambiguities in FTSs (in Sect. 5); illustrate the application of the algorithm on FTSs from the literature (in Sect. 6); discuss the usefulness of removing ambiguities from FTSs (in Sect. 7) and scalability issues of our approach (in Sect. 8); after which we conclude by briefly outlining some planned future work (in Sect. 9).

2 BACKGROUND

In this section, we provide some background needed for the sequel. We recall LTSs as the underlying behavioural structure of FTSs.

Definition 2.1 (LTS). A *Labelled Transition System* (LTS) is a quadruple (S, Σ, s_0, δ), 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 we will use in the paper.

Definition 2.2 (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 $i > 0$.

FTSs were introduced in [30, 32] to concisely model the behaviour of all the products of an SPL in one transition system by annotating transitions with conditions expressing their existence in products. Let $\mathbb{B} = \{\top, \bot\}$ denote the Boolean constants true ($\top$) and false ($\bot$), and let $\mathbb{B}(F)$ denote the set of Boolean expressions over a set of features F (i.e., using features as propositional variables). We do not formalise a language for Boolean expressions in order to allow the inclusion of all possible propositional connectives and, 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 2.3 (FTS). A *Featured Transition System* (FTS) is a sextuple $(S, \Sigma, s_0, \delta, F, \Lambda)$, 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*. Without loss of generality, we assume that whenever $(s, a, \phi, s'), (s, a, \phi', s') \in \delta$, then $\phi = \phi'$ (*transition injectivity*).

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

The notions from Definition 2.2 (path, reachability, deadlock) are carried over to FTSs by ignoring the feature expressions. The transition injectivity in Definition 2.3, guaranteeing that a transition identifies a unique feature expression (as in [21, 62]), turns out to be useful for some of the technical results in this paper. Moreover, we know from [30] (Theorem 8) that restricting feature expressions to singleton features does not affect expressiveness.

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

Definition 2.4 (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. The set of products of $\mathcal{F}$ is denoted by $\mathrm{lts}(\mathcal{F})$.

Note that, by construction: (*i*) each product does not contain unreachable states or transitions; (*ii*) each must transition of the FTS is maintained in the products in which it is reachable; and (*iii*) each product does not contain states or actions that were not originally present in the FTS.

Let $\mathrm{FM}_\mathcal{F}$ denote the *feature model expression* of $\mathcal{F}$, i.e., a feature expression that represents Λ (like, e.g., the formula, in conjunctive normal form, $\bigvee_{\lambda \in \Lambda} (\bigwedge_{f \in F} (\{\, f \mid \lambda(f) = \top \,\} \cup \{\, \neg f \mid \lambda(f) = \bot \,\}))$). We may write FM instead of $\mathrm{FM}_\mathcal{F}$ if no confusion can arise.

Example 2.5. In Fig. 1, we depict an FTS $\mathcal{F}$, modelling the behaviour of a product line of coffee machines, adapted from [10].

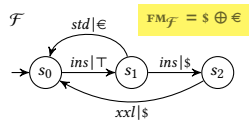

Figure 1: FTS of a product line of coffee machines

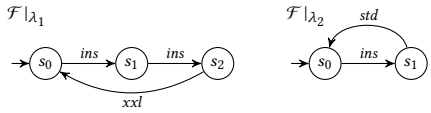

Figure 2: LTSs of products λ_1 and λ_2 of the FTS of Fig. 1

This FTS 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 can thus be represented by the feature expression FM$_{\mathcal{F}}$ = $ ⊕ €, where ⊕ denotes the *exclusive disjunction* operation. Hence the product configurations of $\mathcal{F}$ are $\Lambda = \{\lambda_1, \lambda_2\}$, where $\lambda_1(\$) = \top$, $\lambda_1(€) = \bot$, $\lambda_2(\$) = \bot$, and $\lambda_2(€) = \top$.

The FTS has actions to insert coins (*ins*) and to pour standard (*std*) or extra large (*xxl*) coffee. Extra large coffee is exclusively available for the American market (for two dollars), while standard coffee is exclusively available for the European market (for one euro). The LTSs $\mathcal{F}|_{\lambda_1}$ and $\mathcal{F}|_{\lambda_2}$, depicted in Fig. 2, model the behaviour of the two products of $\mathcal{F}$: configuration λ_1 for the American market and configuration λ_2 for the European market.

3 AMBIGUITIES IN FTSs

Some of the better known analysis operations that are typically performed as part of the automated analysis of feature models concern the detection of anomalies (cf., e.g., [20, 61]). 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 of the products of the SPL, typically due to an incorrect use of cross-tree constraints, whereas a feature is *false optional* if it is contained in all the products of the SPL even though it is not a designated mandatory feature.

In this section, we capture equivalent notions in a behavioural setting, by adapting them to (featured) transitions of an FTS: we define ambiguous FTSs (Sect. 3.1) and show how to transform an ambiguous FTS into an unambiguous one (Sect. 3.2).

3.1 Behavioural Ambiguities

Definition 3.1 (dead transition). We define a transition (of an FTS) to be *dead* if it is not reachable in any of the FTS' products.

Definition 3.2 (false optional transition). A featured transition (of an FTS) is *false optional* if: (i) it is not annotated with the feature expression $\top$ and (ii) it is present in all the FTS' products in which its source state is reachable.

An important safety property of reactive 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, 56]. In case of configurable reactive systems, like SPLs, this notion can be extended to guaranteeing liveness for each system variant (product). Recall from Sect. 2 that a state of an FTS is said to be a deadlock if it has no outgoing transitions and from Definition 2.4 that all states of a product (LTS) of an FTS are reachable.

Definition 3.3 (hidden-deadlock state). We define a state (of an FTS) to be a *hidden deadlock* if: (i) it is not a deadlock in the FTS and (ii) it is a deadlock in one or more of the FTS' products (LTSs).

Definition 3.4 (ambiguous FTS). An FTS is said to be *ambiguous* if any of its states is a hidden deadlock or if any of its transitions is dead or false optional.

Example 3.5. In Fig. 3(left), we depict an FTS $\mathcal{F}$ with features f_1 and f_2 and feature model FM = $f_1 \oplus f_2$.

The LTSs $\mathcal{F}|_{\lambda_1}$ and $\mathcal{F}|_{\lambda_2}$, depicted in Fig. 4(left and middle), model the behaviour of its two valid product configurations: $\lambda_1 = \{f_1\}$ and $\lambda_2 = \{f_2\}$. We immediately see that featured transition $s_2 \xrightarrow{a \mid f_2} s_2$ is dead, featured transition $s_1 \xrightarrow{a \mid f_1} s_2$ is false optional, and state s_2 is a hidden deadlock. Hence $\mathcal{F}$ is an ambiguous FTS.

3.2 Removing Ambiguities

Any ambiguous FTS can be straightforwardly turned into an unambiguous FTS by the following transformation:

(1) remove its dead transitions;
(2) turn its false optional transitions into must transitions; and
(3) make its hidden deadlocks explicit by adding to Q a distinguished deadlock state $s_\dagger \notin Q$ 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 featured transitions in step (1).

Example 3.6. In Fig. 5(left), we depict an unambiguous FTS $\mathcal{F}_*$ that was obtained by transforming the ambiguous FTS $\mathcal{F}$ of Fig. 3. We removed dead featured transition $s_2 \xrightarrow{a \mid f_2} s_2$ and false optional featured transition $s_1 \xrightarrow{a \mid f_1} s_2$ was turned into must transition $s_1 \xrightarrow{a \mid \top} s_2$. Note that there was no need to add an explicit deadlock transition from the hidden deadlock state s_2 to a newly added explicit deadlock state, since s_2 became an explicit deadlock state upon removing the dead featured transition $s_2 \xrightarrow{a \mid f_2} s_2$.

Now consider the ambiguous FTS $\mathcal{F}'$ depicted in Fig. 3(right) with features f_1 and f_2 and feature model FM = $f_1 \oplus f_2$. The LTSs $\mathcal{F}'|_{\lambda_1}$ and $\mathcal{F}'|_{\lambda_2}$, depicted in Fig. 4(left and right), model the behaviour of its two valid product configurations: $\lambda_1 = \{f_1\}$ and $\lambda_2 = \{f_2\}$. Similar to Example 3.5, featured transition $s_2 \xrightarrow{a \mid f_2} s_2$ is dead. However, featured transition $s_1 \xrightarrow{a \mid f_1} s_2$ 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 the same reason as before) but so is state s_1, since it is a deadlock in $\mathcal{F}'|_{\lambda_2}$. Hence also $\mathcal{F}'$ is ambiguous.

Figure 3: Ambiguous FTSs

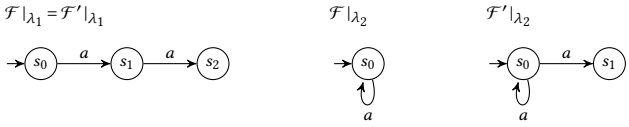

Figure 4: LTSs of products λ_1 and λ_2 of the FTSs of Fig. 3

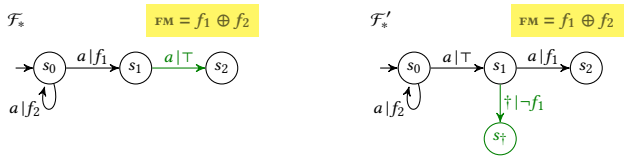

Figure 5: Unambiguous FTSs obtained from the FTSs of Fig. 3

In Fig. 5(right), we depict an unambiguous FTS $\mathcal{F}'_*$ that was obtained by transforming the ambiguous FTS $\mathcal{F}'$ of Fig. 3 as follows. We removed the dead featured transition $s_2 \xrightarrow{a\,|\,f_2} s_2$ and we added the explicit 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 explicit deadlock transition, state s_1 would remain a hidden deadlock state in $\mathcal{F}'_*$.

4 CRITERIA FOR AMBIGUITIES

The ambiguities in FTSs can be characterised by criteria that enable the definition of the static analysis algorithm given in Sect. 5.

Our goal is to spot ambiguities by exploring once the whole FTS in order to infer properties holding for all its products. It is worth noting that the reachability of a state in an FTS (via a path) is not a sufficient condition to ensure that there exist a product including such a state (because the conjunction of the feature expressions of the transitions in each path reaching the state may be false for all configurations). Anyway, the set of configurations of the FTS contains sufficient information to identify its products. In order to circumvent the generation of all products and analyse them one by one, we use Boolean formulas suitable for SAT solver processing.

Definition 4.1 (FTS notations). Let $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$ be an FTS.

(1) For a transition $t = (s, a, \phi, s') \in \delta$, we denote its source state s by $t.\textsc{source}$, its target state s' by $t.\textsc{target}$, and its feature expression ϕ by $t.\textsc{bx}$.
(2) We write $\textsc{Paths}(s)$ to denote the set of all (finite) paths (starting from the initial state s_0 and) ending in state $s \in S$, and we write $\textsc{Paths}(t)$ to denote the set of all (finite) paths (starting from the initial state s_0 and) ending with transition $t \in \delta$.
(3) For a path p, we define its *path expression*, denoted by $\textsc{bx}_p$, as the formula $\bigwedge_{t \in p}(t.\textsc{bx})$, i.e., the conjunction of all formulas labelling the transitions visited by p.

Finiteness of an FTS does not imply that the set $\textsc{Paths}(s)$ is finite. For instance, the FTS $\mathcal{F}$ of Fig. 1 is such that $\textsc{Paths}(s_0)$ contains an infinite number of paths of the form $s_0 t_1 s_1 t_2 s_0 \ldots s_0 t_1 s_1 t_2 s_0$, where $t_1 = (s_0, ins|\top, s_1)$ and $t_2 = (s_1, std|\in, s_0)$.

Definition 4.2 (cycle-free path). Let $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$ be an FTS. A path is *cycle-free* whenever it visits each state at most once. Let $\textsc{cfPaths}(s)$ denote the set of all cycle-free paths ending in state $s \in S$. Let $\textsc{cfPaths}(t)$ denote the set of all cycle-free paths ending with transition $t \in \delta$.

Since a cycle-free path visits each transition at most once, finiteness of an FTS ensures that $\textsc{cfPaths}(t)$ is finite. A cycle-free path p can be seen as the canonical representative of the equivalence class of the paths that become p when removing the cycles.

Lemma 4.3. *Let $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$ be an FTS and let $s \in S$. If $p \in \textsc{Paths}(s)$, then the path $q \in \textsc{cfPaths}(s)$ obtained from p by removing its cycles is such that $\textsc{bx}_p \Rightarrow \textsc{bx}_q$.*

Proof. By construction, for all transitions t it holds that $t \in q$ implies $t \in p$. Thus, by Definition 4.1, $\textsc{bx}_p \Rightarrow \textsc{bx}_q$ holds. □

4.1 Dead Transitions Criterion

The next lemma formalises Definition 3.1 in terms of an FTS' paths, explicating that to decide whether a transition is dead it suffices to inspect the path expressions of all paths ending with that transition. Then, Lemma 4.5 states that it suffices to consider cycle-free paths.

Lemma 4.4. *Let $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$ be an FTS, let $t \in \delta$, and let $\textsc{Paths}(t) \neq \emptyset$. The transition t is dead if and only if $\textsc{fm}_\mathcal{F} \wedge \textsc{bx}_p$ is not satisfiable, for all $p \in \textsc{Paths}(t)$.*

Proof. ($\Rightarrow$) We prove the contrapositive. Let $p^* \in \textsc{Paths}(t)$ be the path $s_0 t_1 s_1 \ldots s_{n-1} t_n s_n$ such that $\textsc{fm} \wedge \textsc{bx}_{p^*}$ is satisfiable. A formula is satisfiable whenever it can be made true by assigning appropriate logical values to its variables; namely, there exists a configuration λ^* that assigns logical values to features such that $\lambda^* \models \textsc{fm} \wedge \textsc{bx}_{p^*}$; namely, both $\lambda^* \models \textsc{fm}$ and $\lambda^* \models \textsc{bx}_{p^*}$. But $\lambda^* \models \textsc{fm}$ simply means that $\lambda^* \in \Lambda$. Moreover, $\lambda^* \models \textsc{bx}_{p^*}$ means that, for all $1 \leq i \leq n$, if $t_i = (s_{i-1}, a_i, \phi_i, s_i)$, then $\lambda^*(\phi_i) = \top$; this is trivial in case $\phi_i = \top$. Concluding, p^* identifies a path reaching t in the product $\mathcal{F}|_{\lambda^*}$ defined by λ^*, so t is not dead and the proof is done.

($\Leftarrow$) We prove the contrapositive. Let $t = (s, a, \phi, s')$ and assume that t is not dead, i.e., $t' = (s, a, s')$ is reachable in a product $\mathcal{F}|_{\lambda^*}$ defined by the configuration $\lambda^* \in \Lambda$. Namely, t' is visited by a path $p = s_0 t'_1 s_1 t'_2 s_2 \ldots s_{n-1} t'_n s_n$ of $\mathcal{F}|_{\lambda^*}$ such that $s = s_{n-1}$, $s' = s_n$, and $t' = t'_n$, for some $n > 0$. Then, the transition injectivity (cf. Definition 2.3) guarantees there exists a unique list of transitions $t_1, \ldots, t_n \in \delta$ such that $t_n = t$ and, for all $1 \leq i \leq n$, $t_i = (s_{i-1}, a_i, \phi_i, s_i)$ for some ϕ_i such that $\lambda^*(\phi_i) = \top$. Thus the path $s_0 t_1 s_1 \ldots s_{n-1} t_n s_n$ is included in $\textsc{Paths}(t)$ by Definition 4.2 and $\lambda^* \models \textsc{bx}_p$. As $\lambda^* \models \textsc{fm}$, we conclude that $\textsc{fm} \wedge \textsc{bx}_p$ is satisfiable. □

Lemma 4.5. *$\textsc{fm} \wedge \textsc{bx}_p$ is not satisfiable, for all $p \in \textsc{Paths}(t)$, if and only if $\textsc{fm} \wedge \textsc{bx}_q$ is not satisfiable, for all $q \in \textsc{cfPaths}(t)$.*

Proof. Direction ($\Rightarrow$) is trivial, because $\textsc{cfPaths}(t) \subseteq \textsc{Paths}(t)$. Direction ($\Leftarrow$) follows by Lemma 4.3, we prove the contrapositive. If there is a path $p \in \textsc{Paths}(t)$ such that $\textsc{fm} \wedge \textsc{bx}_p$ is satisfiable, then we can find a path $q \in \textsc{cfPaths}(t)$ such that $\textsc{fm} \wedge \textsc{bx}_q$ is satisfiable. □

The next theorem, which follows by Lemmas 4.4 and 4.5, provides an equivalent definition that can be used as an effective criterion to decide whether a transition is dead (cf. Definition 3.1).

Theorem 4.6 (dead transitions criterion). *Let $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$ be an FTS, let $t \in \delta$, and let $cfPaths(t) \neq \emptyset$. The transition t is dead if and only if $FM_{\mathcal{F}} \wedge BX_q$ is not satisfiable, for all $q \in cfPaths(t)$.*

4.2 False Optional Transitions Criterion

The next lemma formalises Definition 3.2 in terms of an FTS' paths, explicating that to decide whether a transition is false optional it suffices to inspect the path expressions of all paths ending in the source state of that transition. Then, Lemma 4.8 states that it suffices to consider cycle-free paths.

Lemma 4.7. *Let $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$ be an FTS and let $t \in \delta$ be such that $t.BX \neq \top$. The transition t is false optional if and only if, for all $p \in Paths(t.source)$, $FM_{\mathcal{F}} \Rightarrow (BX_p \Rightarrow t.BX)$ is a tautology.*

Proof. ($\Rightarrow$) We recall that t is false optional whenever for all $\lambda \in \Lambda$, for all $p \in \text{Paths}(t.\text{source})$, $\lambda \models \text{BX}_p$ implies $\lambda \models t.\text{BX}$. We assume that t is false optional and prove that $\text{FM} \Rightarrow (\text{BX}_p \Rightarrow t.\text{BX})$ is a tautology, for all $p \in \text{Paths}(t.\text{source})$. Let λ^* be an assignment of logical values to all features. We consider two sub-cases. First, assume that $\lambda^* \not\models \text{FM} \wedge \text{BX}_p$. Then immediately $\lambda^* \models \text{FM} \Rightarrow (\text{BX}_p \Rightarrow t.\text{BX})$. Second, assume that $\lambda^* \models \text{FM} \wedge \text{BX}_p$. Clearly $\lambda^* \in \Lambda$, because $\lambda^* \models \text{FM}$. Moreover, $\lambda^* \models t.\text{BX}$ because $\lambda^* \models \text{BX}_p$ and t is false optional. Therefore, also in this case we conclude $\lambda^* \models \text{FM} \Rightarrow (\text{BX}_p \Rightarrow t.\text{BX})$. Summarising, $\text{FM} \Rightarrow (\text{BX}_p \Rightarrow t.\text{BX})$ is a tautology.

($\Leftarrow$) Assume that $\text{FM} \Rightarrow (\text{BX}_p \Rightarrow t.\text{BX})$ is a tautology, for all $p \in \text{Paths}(t.\text{source})$. If, for all $p \in \text{Paths}(t.\text{source})$, for all $\lambda^* \in \Lambda$, $\lambda \not\models \text{BX}_p$, then t is trivially false optional. Hence, we assume that $\lambda^* \in \Lambda$ is such that $\lambda^* \models \text{BX}_{p^*}$ for a $p^* \in \text{Paths}(t.\text{source})$ and we aim to prove that $\lambda^* \models t.\text{BX}$. Clearly, $\lambda^* \models \text{FM}$ by Definition 4.1. Therefore $\lambda^* \models \text{FM} \Rightarrow (\text{BX}_p \Rightarrow t.\text{BX})$ implies $\lambda^* \models t.\text{BX}$, which completes the proof. □

Lemma 4.8. *$FM \Rightarrow (BX_p \Rightarrow t.BX)$ is a tautology, for all $p \in Paths(t.source)$, if and only if $FM \Rightarrow (BX_q \Rightarrow t.BX)$ is a tautology, for all $q \in cfPaths(t.source)$.*

Proof. Direction ($\Rightarrow$) is trivial, because $\text{cfPaths}(s) \subseteq \text{Paths}(s)$. Direction ($\Leftarrow$) is proved by contraposition. Let λ^* be an assignment of logical values to all features such that $\lambda^* \not\models \text{FM} \Rightarrow (\text{BX}_p \Rightarrow t.\text{BX})$, for some $p \in \text{Paths}(t.\text{source})$. This means that $\lambda^* \models \text{FM} \wedge \text{BX}_p$ and $\lambda^* \not\models t.\text{BX}$. By Lemma 4.3, there is a path $q \in \text{cfPaths}(t)$ such that $\lambda^* \models \text{FM} \wedge \text{BX}_q$ but, still, $\lambda^* \not\models t.\text{BX}$. This proves the logical equivalence. □

The next theorem, which follows by Lemmas 4.7 and 4.8, provides an equivalent definition that can be used as an effective criterion to decide whether a transition is false optional (cf. Definition 3.2).

Theorem 4.9 (false optional transitions criterion). *Let $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$ be an FTS and let $t \in \delta$ be such that $t.BX \neq \top$. The transition t is false optional if and only if, for all $p \in cfPaths(t.source)$, $FM_{\mathcal{F}} \Rightarrow (BX_p \Rightarrow t.BX)$ is a tautology.*

4.3 Hidden Deadlock States Criterion

The next lemma formalises Definition 3.3 in terms of an FTS' paths, explicating that to decide whether a state is a hidden deadlock it suffices to inspect the path expressions of all paths ending with a transition having such state as its source state. Then, Lemma 4.11 states that it suffices to consider cycle-free paths.

Lemma 4.10. *Let $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$ be an FTS, let $s \in S$, and let $s.EXIT\_TRS$ denote the set of transitions starting from s. The state s is **not** a hidden deadlock if and only if, either $s.EXIT\_TRS = \emptyset$ or for all $p \in Paths(s)$, $FM_{\mathcal{F}} \Rightarrow (BX_p \Rightarrow (\bigvee_{t \in s.EXIT\_TRS} t.BX))$ is a tautology.*

Proof. ($\Rightarrow$) A state s is not a hidden deadlock whenever, either $s.EXIT\_TRS = \emptyset$ or for all configurations $\lambda \in \Lambda$ and for all paths $p \in \text{Paths}(s)$, $\lambda \models \text{BX}_p$ implies that there exists a transition $t \in s.EXIT\_TRS$ such that $\lambda \models t.\text{BX}$. Assume that s is not a hidden deadlock. If $s.EXIT\_TRS = \emptyset$, then the proof is immediate; hence we assume that for all $\lambda \in \Lambda$ and for all $p \in \text{Paths}(s)$, $\lambda \models \text{BX}_p$ implies $\lambda \models (\bigvee_{t \in s.EXIT\_TRS} t.\text{BX})$ and we aim to prove that $\text{FM} \Rightarrow (\text{BX}_p \Rightarrow \bigvee_{t \in s.EXIT\_TRS} t.\text{BX})$ is a tautology. The proof is straightforward, because for all $\lambda \in \Lambda$ we know that $\lambda \models \text{FM}$.

($\Leftarrow$) If s is such that $s.EXIT\_TRS = \emptyset$, then the proof is immediate. Let $p \in \text{Paths}(s)$ be such that $\text{FM} \Rightarrow (\text{BX}_p \Rightarrow (\bigvee_{t \in s.EXIT\_TRS} t.\text{BX}))$ is a tautology. This means that for all assignments λ of logical values to features, the following holds: if $\lambda \models \text{FM}$ and $\lambda \models \text{BX}_p$, then $\lambda \models (\bigvee_{t \in s.EXIT\_TRS} t.\text{BX})$. Clearly, $\lambda \models \text{FM}$ implies $\lambda \in \Lambda$, so the proof is straightforward. □

Lemma 4.11. *$FM \Rightarrow (BX_p \Rightarrow (\bigvee_{t \in s.EXIT\_TRS} t.BX))$ is a tautology, for all $p \in Paths(s)$, if and only if $FM \Rightarrow (BX_q \Rightarrow (\bigvee_{t \in s.EXIT\_TRS} t.BX))$ is a tautology, for all $q \in cfPaths(s)$.*

Proof. Direction ($\Rightarrow$) is trivial, because $\text{cfPaths}(s) \subseteq \text{Paths}(s)$. Direction ($\Leftarrow$) is proved by contraposition. Let λ^* be an assignment of logical values to all features such that $\lambda^* \not\models \text{FM} \Rightarrow (\text{BX}_p \Rightarrow (\bigvee_{t \in s.EXIT\_TRS} t.\text{BX}))$, for some path $p \in \text{Paths}(s)$. This means that $\lambda^* \models \text{FM} \wedge \text{BX}_p$ and $\lambda^* \not\models (\bigvee_{t \in s.EXIT\_TRS} t.\text{BX})$. By Lemma 4.3, there is a path $q \in \text{cfPaths}(s)$ such that $\lambda^* \models \text{FM} \wedge \text{BX}_q$ but, still, $\lambda^* \not\models (\bigvee_{t \in s.EXIT\_TRS} t.\text{BX})$. This proves the logical equivalence. □

The next theorem, which follows by Lemmas 4.10 and 4.11, provides an equivalent definition that can be used as an effective criterion to decide whether a state is a hidden deadlock (cf. Definition 3.3).

Theorem 4.12 (hidden deadlock states criterion). *Let $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$ be an FTS, let $s \in S$, and let $s.EXIT\_TRS$ denote the set of transitions starting from s. The state s is a hidden deadlock if and only if both $s.EXIT\_TRS \neq \emptyset$ and there exists a path $p \in cfPaths(s)$ such that $FM_{\mathcal{F}} \Rightarrow (BX_p \Rightarrow (\bigvee_{t \in s.EXIT\_TRS} t.BX))$ is **not** a tautology.*

4.4 SAT Solving

All criteria presented in this section are defined as deciding for a given Boolean formula if it is a tautology or not satisfiable. Checking these criteria are thus variations of the SAT problem [33], where instead of finding if a Boolean formula has a solution, we want to find if it has no solution (e.g., a formula ϕ is a tautology iff $\neg\phi$

has no solution). We can thus remark that these problems are co-NP-complete. Additionally, SAT solvers can be used to solve these problems. SAT solving is an active field of research [4, 22, 48, 50] 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 of a criterion to a SAT solver and checking if it fails, we can conclude if the criterion is validated or not. In our implementation, we used the Z3 SMT solver [40, 57] (that includes a SAT solver) developed by Microsoft Research and freely available under the MIT license.

5 STATIC ANALYSIS ALGORITHM

In this section, we present an algorithm that visits all cycle-free paths (starting from the initial state) of an FTS in a depth-first order. We describe the algorithm via python pseudocode that identifies the bodies of functions, selection, and iteration operators, by means of a suitable indentation, in place of the more common C-like curly brackets (cf. Listings 1 and 2). The algorithm assumes that as input is provided a suitable representation of an FTS. Then in a *unique* FTS traversal it is able to identify all ambiguities: hidden deadlock states (cf. Definition 3.3) and dead and false optional transitions (cf. Definitions 3.1 and 3.2). We remark that our solution rests on the use of a suitable SAT solver.

Listing 1: Ambiguities discovery algorithm

```
for t in fts.trs: // initialise transition deadness & optionality
   t.dead ←true
   if (t.bx ≠true): t.false_opt == true
   else: t.false_opt == false
for s in fts.states: s.live ←true // initialise state liveness
path_discover(fts.initial, true)
for s in fts.states:            // set state hidden deadlockness
   s.hDead ←(s.exit_trs ≠ ∅ && not (s.live))
```

Listing 2: Depth-first visit procedure

```
def path_discover(s, bx_p):
  s.visited ←true
  if(s.live == true && (s.exit_trs ≠ ∅)):
     s.live ←((fts.fm ⇒ (bx_p ⇒ (⋁_{t∈s.exit_trs} t.bx))) is tautology)
  for t ∈ s.exit_trs:
     if(t.false_opt == true)
        t.false_opt ←((fts.fm ⇒ (bx_p ⇒ t.bx)) is tautology)
     ext_bx_p ←bx_p && t.bx
     valid_p ←((fts.fm && ext_bx_p) is satisfiable)
     if(t.dead == true): t.dead ←not(valid_p)
     if((not t.exit_state.visited) && valid_p):
        path_discover(t.exit_state, ext_bx_p)
  s.visited ←false
```

First of all, we introduce the structure of global variables. The considered FTS is stored in the global variable `fts`. It has four (relevant) fields:

(1) `trs` is the set of all the transitions in the FTS;
(2) `states` is the set of all states in the FTS;
(3) `initial` is the initial state of the FTS;
(4) `fm` is the Boolean expression representing the feature model of the FTS.

The structure of `states` has four fields as well:

(1) `visited` is a Boolean value indicating if the state is included in the path currently visited;
(2) `exit_trs` is the set of transitions exiting from this state;
(3) `live` is a Boolean flag that stores the discovery of products with states (having at least one outgoing transition) that are deadlocks;
(4) `hDead` is a Boolean flag meant to mark the hidden deadlock status of states.

The data structure of transitions is crucial:

(1) `bx` is a Boolean expression formalising the FTS logic constraint on features that identifies in which variants the transition has to be included;
(2) `exit_state` is the state to which the transition points;
(3) `dead` is a Boolean flag meant to mark the dead status of transitions;
(4) `false_opt` is a Boolean flag meant to mark the false optional status of transitions.

It is worth to recall that, in accordance with Definition 2.2, we are interested only in paths starting in the initial state of the FTS; moreover, we focus on cycle-free paths that never cross twice the same state or the same transition.

Definition 5.1 (path notations). Let $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$ be an FTS represented in above data structures and let $p = s_0 t_1 s_1 \dots s_{n-1} t_n s_n$ ($n \in \mathbb{N}$) be a cycle-free path of $\mathcal{F}$.

- Let p.endState denote the final state of p, namely s_n.
- Let p.endTrs denote the final transition of p, namely t_n.
- Let VsTed(p) denote the predicate that holds whenever: s.visited == **true** iff $s = s_i$, for all $i \leq n-1$.
- Let SaTed(p) denote the predicate that holds exactly when fts.fm $\wedge$ bx_p is satisfiable.
- Let EXT_p denote the set of all cycle-free paths extending p in a product, i.e., all paths $p' = s_0 t_1 s_1 \dots s_{n-1} t_n s_n \dots t_{n+k} s_{n+k}$ ($k \geq 0$) such that SaTed(p') holds.

We remark that VsTed(p) predicates that the states marked as "visited" are exactly those in p, except for the last one. Moreover, as stated by the following proposition, EXT_p is closed under prefixes extending p.

Proposition 5.2 (ext-closure). *Let $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$ be an FTS and let p be one of its cycle-free path. If $p' \in \mathrm{EXT}_p$ then $\mathrm{EXT}_{p'} \subseteq \mathrm{EXT}_p$.*

Proof. If SaTed(p) is false then $\mathrm{EXT}_{p'} = \mathrm{EXT}_p = \emptyset$. If there is $p'' \in \mathrm{EXT}_{p'}$ then p'' extends p' and SaTed(p'') holds. Clearly p'' extends also p, so we conclude $p'' \in \mathrm{EXT}_p$. □

We first prove correctness of the procedure `path_discover` from Listing 2.

Lemma 5.3 (correctness of procedure `path_discover`). *Let $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$ be the FTS represented in the algorithm data structures and let $p = s_0 t_1 s_1 \dots s_{n-1} t_n s_n$ ($n \in \mathbb{N}$) be a cycle-free path such that both SaTed(p) and VsTed(p).*

(1) *The execution of* path_discover(p.endState, bx_p) *recursively calls the function* path_discover(p'.endState, $\mathrm{bx}_{p'}$) *in a situation in which VsTed(p') holds,* ***for all and only*** *$p' \in \mathrm{EXT}_p$ and $p \neq p'$.*
(2) *The execution of* path_discover(fts.initial, **true**) *always ends in a finite number of steps.*

(3) *Let* $p' \in \textsc{ext}_p$ *and* $s \in p'.\textsc{endState}$. *If* s.exit_trs $\neq \emptyset$ *and* $\mathit{FM}_{\mathcal{F}} \Rightarrow (\mathit{BX}_p \Rightarrow (\bigvee_{t \in s.\mathit{EXIT\_TRS}} t.\mathit{BX}))$ *is not a tautology, then the execution of* path_ discover(p.endState, bx_p) *ensures that* s.live *is flagged false.*

(4) *Let* $p' \in \textsc{ext}_p$, $s \in p'.\textsc{endState}$, *and* $t \in$s.exit_trs. *If* $\mathit{FM}_{\mathcal{F}} \Rightarrow (\mathit{BX}_{p'} \Rightarrow t.\mathit{BX})$ *is not a tautology, then the execution of* path_discover(p.endState, bx_p) *ensures that* t.false_opt *is flagged false.*

(5) *Let* $p' \in \textsc{ext}_p$ *be a path including at least a transition, and let* t *be* $p'.\textsc{endTrs}$. *The execution of* path_discover(p.endState, bx_p) *ensures that* t.dead *is flagged false.*

Proof. (1) First we show that, if $p' \in \textsc{ext}_p$, this implies a recursive call to path_discover(p'.endState, $\mathrm{bx}_{p'}$) (in a situation in which VsTed(p') holds). The proof is by induction on the number of paths in $\textsc{ext}_p$. If $\textsc{ext}_p = \emptyset$, the proof is trivial; so, assume $p' = s_0 t_1 s_1 \ldots s_{n-1} t_n s_n t_{n+1} s_{n+1} \in \textsc{ext}_p$ extending p with the unique additional transition t_{n+1}. Execution of path_discover(p.endState, bx_p) is driven by the code of Listing 2. Line 2 predisposes a flag visited and makes VsTed(p') hold, thus a first statement requirement is satisfied. Then, since p' is cycle-free and SaTed(p') holds, lines 11–12 concretely do the recursive call path_discover(p'.endState, $\mathrm{bx}_{p'}$). As $\textsc{ext}_{p'} \subseteq \textsc{ext}_p$ by Proposition 5.2, the statement holds for all paths in $\textsc{ext}_{p'}$ by induction; so, the proof follows.
Assume that, by executing path_discover(p.endState, bx_p) a recursive call to path_discover(p'.endState, $\mathrm{bx}_{p'}$) is done, in a situation in which VsTed(p') holds. The proof is done by induction on the number N of transitions of p' not in p. The call must be done by means of the lines 11–12 of Listing 2. If $N = 1$ then line 11 ensures that p' is still a cycle-free path (properly extending p) and SaTed(p'); namely $p' \in \textsc{ext}_p$. If $N \geq 1$ then the proof follows by induction.

(2) As usual, we define FTSs without finiteness assumptions; but as for SPLs, their practical interest implicitly assumes that all constituents of $\mathcal{F}$ are finite (this allows to store $\mathcal{F}$ in our algorithm's data structure). First note that the number of transitions starting from the nodes of the FTS is bounded by $\max_{s \in S} \#(s.\textsc{exit\_trs}) \in \mathbb{N}$. Thus, line 5 of the procedure in Listing 2 does a finite number of recursive calls. Next, note that the number of states visited==**false** is strictly decreased at each of the previous calls. Thus, the proof follows.

(3) If $p \neq p'$ then, path_discover(p'.endState, $\mathrm{bx}_{p'}$) is recursively called by the point 1 of this lemma. Line 3 of the procedure prevents the update of the flag (p'.endState).live whenever it is already set to false. Line 4, if necessary, updates this flag in accordance to our statement.

(4) If $p \neq p'$ then, path_discover(p'.endState, $\mathrm{bx}_{p'}$) is recursively called by the point 1 of this lemma. Line 6 of the procedure prevents the update of the flag (p'.endTrs).false_opt whenever it is already set to false. Line 7, if necessary, updates this flag in accordance to our statement.

(5) $p' \in \textsc{ext}_p$ ensures that SaTed(p'), i.e., p is a path such that fts.fm $\wedge$ bx_p is satisfiable. Line 9 of the procedure stores true in valid_p. Line 10 prevents the update of the flag (p'.endTrs).dead whenever it is already false and, if necessary, updates this flag in accordance to our statement. □

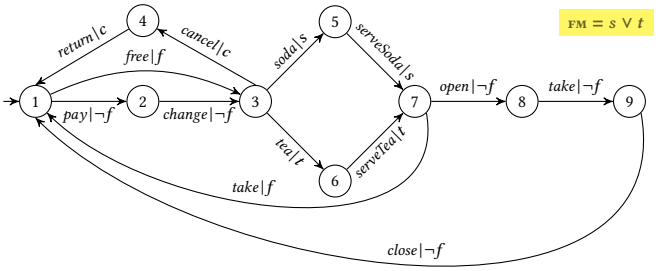

Figure 6: FTS of the vending machine from [27]

Next we prove correctness of the analysis algorithm in Listing 1.

Theorem 5.4 (correctness of algorithm).
Let $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$ *be an FTS represented in the algorithm data structures. The execution of the* ambiguities discovery algorithm *(cf. Listing 1) sets the flags* dead, false_opt, *and* hDead *in the data structure in accordance with the fact that a transition is dead, a transition is false optional, and a state is not a hidden deadlock, respectively.*

Proof. The algorithm of Listing 1 begins by executing some initialisation, after which it calls path_discover(fts.initial, **true**). Lemma 5.3 specifies the content of the flags dead, false_opt, and live following the aforementioned procedure call. The algorithm ends by setting the flags s.hDead as expected. Hence, the proof follows by Theorems 4.6, 4.9 and 4.12. □

6 ILLUSTRATIVE EXAMPLES

The python code allowing the verification of the examples presented in this section and in Sect. 8 is publicly available [7].

In Fig. 6, we depict an example FTS modelling the behaviour of a configurable vending machine selling soda and tea from [27], an FTS benchmark which was used in numerous publications, among which [5, 6, 11, 30, 32, 42–45, 47]. Its feature model can be represented by the formula $s \vee t$ over the 4 features $\{f, c, s, t\}$, thus resulting in 12 products (i.e., $2^4 - 4$, excluding $\emptyset$, $\{f\}$, $\{c\}$, $\{f, c\}$). The FTS of the vending machine contains 9 states and 13 transitions.

Listing 3 reports the result of applying static analysis to this FTS: no dead transitions and no hidden deadlocks, but 6 false optional transitions, viz. $(2, \mathit{change}, \neg f, 3)$, $(4, \mathit{return}, c, 1)$, $(5, \mathit{serveSoda}, s, 7)$, $(6, \mathit{serveTea}, t, 7)$, $(8, \mathit{take}, \neg f, 9)$, and $(9, \mathit{close}, \neg f, 1)$. Hence, the FTS is ambiguous, but it suffices 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 Fig. 6

```
Vending Machine: live
 LIVE STATES = [1,2,3,4,5,6,7,8,9]
 DEAD TRANSITIONS = []
 FALSE OPTIONAL TRANSITIONS = [(2,3),(4,1),(5,7),(6,7),(8,9),(9,1)]
 HIDDEN DEADLOCK STATES = []
```

In Fig. 7, we depict an example FTS modelling the behaviour of the so-called *system* FTS that models the logic of a configurable controller of the mine pump model from [26, 27], a standard SPL benchmark for FTSs which was used in numerous publications, among which [10, 28, 30, 32, 35, 36, 43, 45–47]. The *system* FTS of this mine pump model contains 25 states and 41 transitions.[1]

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

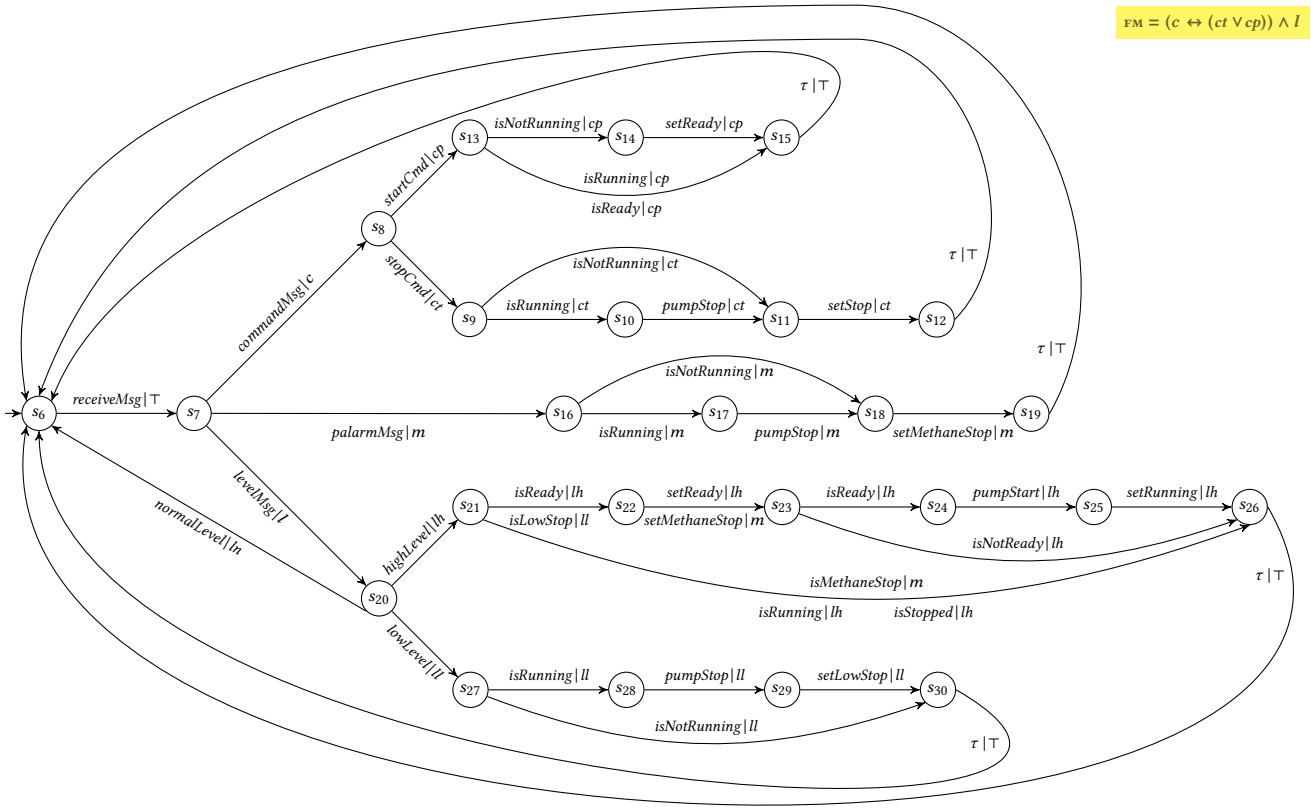

Figure 7: The *system* FTS of the mine pump model from [27] (we have labelled transitions that were unlabelled with $\tau|\top$)

The controller of this mine pump model is the parallel composition of the *system* FTS with the so-called *state* FTS, depicted in Fig. 8.

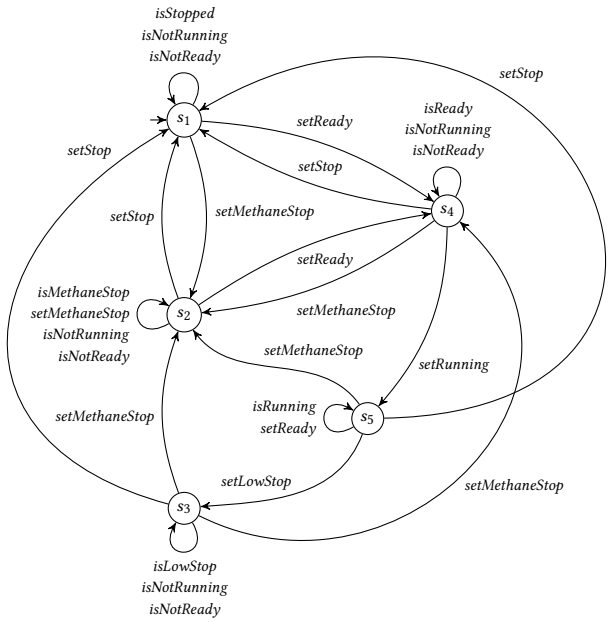

Figure 8: The *state* FTS of the mine pump model from [27]

The mine pump has to keep a mine safe from flooding by pumping water from a shaft while avoiding a methane explosion. The reason we use the model from [26, 27] is that it comes with detailed FTSs. Its feature 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 (i.e., 2^6, since ϕ is equivalent to considering features $\{ct, cp, m, ll, ln, lh\}$ to be optional).

Listing 4 reports the result of applying static analysis to the *system* FTS: no dead transitions, but numerous false optional transitions, among which $(s_7, levelMsg, l, s_{20})$, 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 $(s_7, levelMsg, l, s_{20})$), while s_{20} is a deadlock in all 8 products that lack any of the features from the subset $\{ll, ln, lh\}$.

Listing 4: Result of the static analysis on the FTS of Fig. 7

```
Mine Pump: not live
 LIVE STATES = [S6,S7,S8,S9,S10,S11,S12,S13,S14,S15,S16,S17,S18,S19,
  S21,S22,S23,S24,S25,S26,S27,S28,S29,S30]
 DEAD TRANSITIONS = []
 FALSE OPTIONAL TRANSITIONS = [(S10,S11),(S11,S12),(S13,S14),
  (S13,S15,isReady),(S13,S15,isRunning),(S14,S15),(S16,S17),
  (S16,S18),(S17,S18),(S18,S19),(S21,S22,isReady),
  (S21,S26,isRunning),(S21,S26,isStopped),(S22,S23,setReady),
  (S23,S24),(S23,S26),(S24,S25),(S25,S26),(S27,S28),(S27,S30),
  (S28,S29),(S29,S30),(S7,S20),(S9,S10),(S9,S11)]
 HIDDEN DEADLOCK STATES = [S20]
```

Hence the *system* FTS is ambiguous, but it suffices 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 is 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 [26, 27, 32], instead, an alternative feature model in which only *c* (and implicitly *ct* and *cp*) and *m* are optional is 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 is to slightly change the FTS itself, to make sure that it contains neither a hidden nor an explicit deadlock state. In this case, it suffices to add one or more transitions to leave state s_{20} in a meaningful way. This is the solution opted for in [10, 30, 45–47], which use the specification in fPromela of the complete mine pump model (see below) as distributed with SNIP [28] (http://projects.info.unamur.be/fts/snip/) and ProVeLines [34] (http://projects.info.unamur.be/fts/provelines/) or translations thereof for mCRL2 [37] (http://www.mcrl2.org/) or VMC [17] (http://fmtlab.isti.cnr.it/vmc/). Basically, three transitions are added to the *system* FTS of Fig. 7 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. $(s_{20}, highLevel, \neg lh, s_6)$, $(s_{20}, lowLevel, \neg ll, s_6)$, and $(s_{20}, normalLevel, \neg ln, s_6)$.[2]

In [26, 27], the controller of the mine pump model, composed of the *system* and *state* FTSs, interacts with an environment: it operates a water pump based on methane and water level sensors, modelled by three further FTSs. The parallel composition of these five FTSs is the above mentioned complete mine pump model. We depict the FTS of the methane level in Fig. 9 and refer to [26, 27] for the remaining FTSs. Moreover, *methaneRise* and *methaneLower* are local actions of this FTS that do not synchronise with any of the other four FTSs. Hence, while the solutions presented above make the *system* FTS of Fig. 7 unambiguous, it is immediately 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 demonstrates the usefulness of analysing component FTSs in isolation. More on this in Sect. 8.

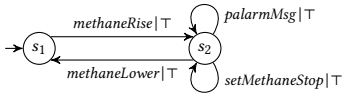

Figure 9: FTS of the methane level environment from [27]

7 USEFULNESS OF UNAMBIGUOUS FTSs

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, the transformation of an ambiguous FTS into an unambiguous FTS also serves another purpose, viz. to facilitate family-based model checking of properties expressed in a fragment of the variability-aware action-based and state-based branching-time modal temporal logic v-ACTL and interpreted on so-called 'live' MTSs [5, 11–13]. 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 [53, 54].

In [12], 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 (LTSs obtained from the MTS in a way similar to Definition 2.4), resulting in an MTS in which every path is infinite. Then it is proved that the validity of formulas expressed in a rich fragment of v-ACTL is preserved in all products (cf. Theorem 4 of [12]), thus allowing 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.[3] Note that any FTS $\mathcal{F}$ can be transformed into an 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 the FTS is unambiguous, then the corresponding MTS is live, with respect to the FTS' set of products $\mathsf{lts}(\mathcal{F})$, because it has no hidden deadlocks, and all transitions of the FTS that are mandatorily present in all products moreover correspond to must transitions in the MTS. This demonstrates that the above mentioned result from [12] can be carried over to unambiguous FTSs, thus allowing family-based model checking of such FTSs for the v-ACTL fragment v-ACTLive$^\Box$. Hence, the following result holds.

Proposition 7.1. *Any formula ϕ of v-ACTLive$^\Box$ is preserved by unambiguous FTSs: given an unambiguous FTS $\mathcal{F}$, whenever $\mathcal{F} \models \phi$, then $\mathcal{F}|_\lambda \models \phi$ for all products $\mathcal{F}|_\lambda \in \mathsf{lts}(\mathcal{F})$.*

In the next section, we apply this result to an example FTS and provide examples of v-ACTLive$^\Box$ formulas to illustrate its impact.

7.1 Family-Based Model Checking

We have seen in Sect. 3.2 how to transform an ambiguous FTS into an unambiguous one. Furthermore, as mentioned above, an FTS can be transformed into an 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. In Fig. 10, we depict the MTS [4] that is obtained in this way from the unambiguous FTS (described in the beginning of Sect. 6) that corresponds to the FTS of Fig. 6.

As we argued in the beginning of this section, the resulting MTS is live, with respect to the FTS' set of products, thus allowing family-based model checking for v-ACTLive$^\Box$ (cf. Proposition 7.1). Example formulas of properties that can now be verified in a family-based manner on the MTS of Fig. 10 are the following:

(1) $AG\, AF_{pay \vee free}\, \top$: *infinitely often, either action pay or action free is executed;*

[2] To satisfy the transition injectivity of Definition 2.3, in the FTS of Fig. 7 this results in the substitution of transition $(s_{20}, normalLevel, ln, s_6)$ with $(s_{20}, normalLevel, \top, s_6)$.

[3] Adding loops (labelled with a dummy symbol) to all final states makes the MTS live.

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

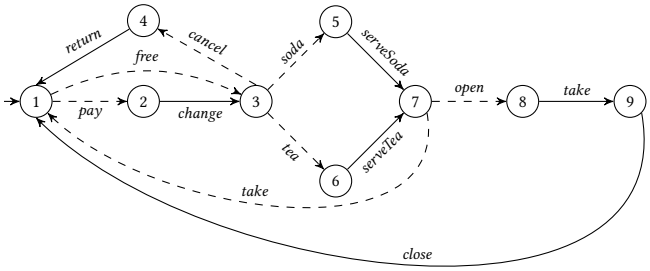

Figure 10: MTS obtained from the FTS of Fig. 6

(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.*[5]

Such type of formulas can efficiently be verified with the variability model checker VMC, which is a tool for the analysis of behavioural SPL models specified as an MTS together with a set of logical variability constraints (akin to feature expressions) [17, 18]. VMC is the most recent member of the KandISTI product line of model checkers developed at ISTI–CNR over the past decades [13, 14]. The KandISTI toolset offers explicit-state on-the-fly model checking of functional properties expressed in specific action-based and state-based branching-time temporal logics derived from ACTL [41], which is the action-based version of the well-known logic CTL [25].

However, it is important to underline that VMC currently uses the variability constraints associated with the MTS to dynamically evaluate the liveness of each node. In this paper, we introduced a static analysis algorithm and transformation to turn any FTS into an unambiguous FTS, and we showed that this allows to establish a priori the liveness of all its nodes, and thus of the corresponding MTS. This makes it possible to improve VMC with the possibility to verify a live MTS without the need to use variability constraints to dynamically evaluate the liveness of its nodes. As a result, VMC could verify the above formulas in a family-based manner on the MTS of Fig. 10, since this MTS is live with respect to the set of products of the FTS of Fig. 6, meaning that whenever a formula holds for the MTS it also holds for all products defined by the FTS.

The static analysis algorithm presented in Sect. 5 and the FTS transformation defined in Sect. 3.2 could also give rise to a new KandISTI tool, tailored for family-based model checking of temporal logic properties on FTSs. At present, efficient SPL model checking against FTSs can be achieved by using dedicated family-based model checkers such as the ProVeLines [34] tool suite (including its predecessor SNIP [28]) or, alternatively, by using one of the highly optimised off-the-shelf classical model checkers such as SPIN or mCRL2, which have recently been made amenable to family-based SPL model checking against FTSs [10, 46].

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

8 SCALABILITY

From our experience, the bottleneck of the static analysis algorithm presented in this paper, as far as computational scalability is concerned, is the identification of all possible cycle-free paths of the FTS under scrutiny. While the static analyses performed on the exemple FTSs reported in Sect. 6 required a negligible amount of time, it is clear that this no longer holds for FTSs with hundreds of states and thousands of transitions. For instance, the FTS of the complete mine pump model of [26, 27], described in Sect. 6, has 417 states and 1255 transitions, which already are too many to efficiently visit all cycle-free paths in a depth-first manner. We plan to investigate optimisations of the static analysis algorithm that allow to reduce the impact of this bottleneck. We also plan to study the resulting algorithm's complexity.

However, it is not very likely that a system of a size like that of the complete mine pump is modelled as one monolithic FTS. Typically, a (large) system is designed in a modular way, as a composition of (smaller) components. A more promising strategy is thus to analyse FTS components in isolation and study the implications that ambiguities detected at the component level have for the entire system. This is confirmed by our analyses of the mine pump system in Sect. 6, where we have seen that the complete mine pump system is deadlock-free (i.e., live) even if the *system* FTS is not, simply because progress is guaranteed by the presence of a cycle of local actions in another component FTS.

In Table 1, we report some hard data concerning static analyses of the FTSs discussed so far. In addition, we report the results for two additional FTSs, viz. the configurable coffee machine from [3] depicted in Fig. 11, which is another FTS benchmark used in numerous publications, among which [8, 9, 12, 15, 16, 18, 19, 21, 26, 62], and the aforementioned controller of the mine pump model from [26, 27], i.e. the parallel composition of the *system* FTS with the *state* FTS.

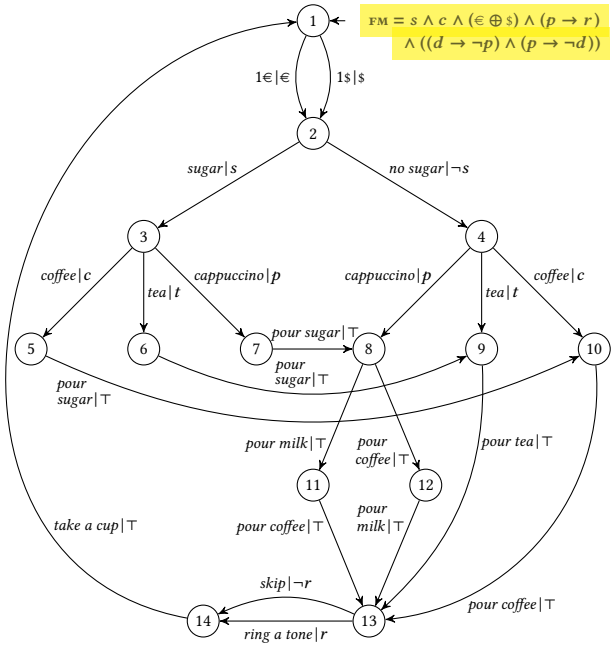

Figure 11: FTS of the coffee machine from [3]

Table 1: Characteristics of the FTSs mentioned in this paper and results of applying static analysis

FTS	characteristics			results of static analysis				computational effort	
Model	# states	# transitions	# actions	live-ness	# dead transitions	# false optional transitions	# hidden deadlock states	run-time (s)	memory use (Mb)
Vending machine [27]	9	13	12	yes	0	6	0	0.68	41
Coffee machine [3]	14	22	14	yes	0	4	0	1.35	42
Mine pump (system) [27]	25	41	22	no	0	25	1	1.41	44
Mine pump (controller) [27]	77	104	22	no	0	59	4	5.37	48
Mine pump (complete) [27]	417	1255	26	yes	?	?	?	timeout	–

The experiments have been performed on a virtual machine [6] with 2048 Mb of allocated memory on a Windows 10 Pro 64 bit with 16 Gb of RAM and a CPU AMD Ryzen 7 1700X (8 core, 16 threads, 3.4 Ghz).

The FTSs of the vending machine (depicted in Fig. 6 and discussed in Sect. 6) and the coffee machine (depicted in Fig. 11) are both live (i.e., no deadlocks), with no dead transitions, while a respective 46% and 18% of their transitions are false optional. Their static analysis is immediate. Also that of the *system* FTS of the mine pump (depicted in Fig. 7 and discussed in Sect. 6) is immediate, but it is not live because one of its 25 states is a hidden deadlock. None of its transitions is dead, but 61% is false optional. Instead, it takes more than 5 s to analyse the FTS of the mine pump controller (i.e., the parallel composition of the *system* FTS and the *state* FTS (depicted in Fig. 8 and discussed in Sect. 6). It is not live, due to 4 hidden deadlock states and 57% of its transitions is false optional. Finally, from the discussion in Sect. 6 we know that the FTS of the complete mine pump, not depicted here, is live, but further results are missing, since we aborted the analysis after several hours.

We have not yet studied in detail what can be said about the preservation of ambiguities in a parallel composition of FTSs, either bottom-up, i.e., from the component FTSs to their parallel composition, or top-down, i.e., from the parallel composition to its constituting component FTSs. We conjecture that applying the static analysis algorithm to individual component FTSs, and consequently removing dead transitions and turning false optional transitions into must transitions, will not affect the behaviour of the parallel composition of these FTSs. On the contrary, it can be shown that in general the parallel composition of two unambiguous FTSs may result in a composed FTS with dead or false optional transitions, and, moreover, that the parallel composition may result in a composed FTS with more or less hidden deadlock states than in the individual component FTSs.

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 also allows more efficient model checking of the composed system (cf. Sect. 7.1). Instead, the application of the static analysis algorithm to a composed FTS resulting from the parallel composition of several FTSs is less desirable for at least two reasons. On one hand, it risks to become problematic due to the exponential growth of the number of paths to be considered. On the other hand, the benefits of detecting ambiguities are greatly reduced because of the lack of a detailed specification of the composed FTS, which is merely a semantic model without a matching syntactic specification. Composed configurable systems can be described as Multi SPLs (MPLs), i.e., sets of interdependent SPLs [49]. It is not clear how to obtain results for composed FTSs by reusing results of analyses performed in isolation on its components, in analogy with recently proposed compositional approaches for analysing MPLs [38, 39, 55].

We conclude this section with two ideas to improve the algorithm, based on the fact that even though the number of paths in a graph can be exponentially larger than its size, several heuristic-based optimisations can be implemented to reduce the complexity of our algorithm in many cases. Firstly, paths must be computed only for transitions whose Boolean formula is not true (i.e., those that could be false optional or that cannot easily be stated as dead), and only for states whose live status is not trivial, i.e., states that have input and output transitions, and none of their output transitions' Boolean formula is true. For FTSs with very simple constraints on their transitions, this could greatly reduce the search space when traversing them. Secondly, checking the ambiguity criteria in a cycle-free FTS can be performed with a linear traversal, following the topological order of the FTS. It could be possible to extend this principle to FTSs with cycles, by using the topological order traversal only on the DAG subgraphs of an FTSs and thus greatly improving the efficiency of our algorithm on FTSs with few cycles.

9 CONCLUSION

In this paper, we have introduced several types of static analysis that can be performed over an FTS, we have given an effective algorithm for these analyses and demonstrated its correctness and completeness, and we have shown a number of example applications. The python code allowing to reproduce the verification of the examples presented Sect. 6 and Sect. 8 is publicly available [7].

In future work we plan to: (i) investigate the improvements suggested in the previous section, and (ii) evolve the prototype into a tool that supports to perform the static analysis of generic FTSs (specified in some standard format), to automatically disambiguate them, and to apply v-ACTL model checking to generic FTSs.

[6] Gentoo 201905, CLI Version VirtualBox (VDI) 64 bit, https://www.osboxes.org/gentoo/

ACKNOWLEDGMENTS

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

REFERENCES

[1] Bowen Alpern and Fred B. Schneider. 1985. Defining liveness. *Inf. Process. Lett.* 4, 21 (1985), 181–185. https://doi.org/10.1016/0020-0190(85)90056-0

[2] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer. https://doi.org/10.1007/978-3-642-37521-7

[3] Patrizia Asirelli, Maurice H. ter Beek, Alessandro Fantechi, and Stefania Gnesi. 2011. Formal Description of Variability in Product Families. In *Proceedings of the 15th International Software Product Lines Conference (SPLC'11).* IEEE, 130–139. https://doi.org/10.1109/SPLC.2011.34

[4] Gilles Audemard, Jean-Marie Lagniez, Nicolas Szczepanski, and Sébastien Tabary. 2016. An Adaptive Parallel SAT Solver. In *Proceedings of the 22nd International Conference onPrinciples and Practice of Constraint Programming (CP'16) (LNCS)*, M. Rueher (Ed.), Vol. 9892. Springer, 30–48. https://doi.org/10.1007/978-3-319-44953-1_3

[5] Maurice H. ter Beek, Ferruccio Damiani, Stefania Gnesi, Franco Mazzanti, and Luca Paolini. 2015. From Featured Transition Systems to Modal Transition Systems with Variability Constraints. In *Proceedings of the 13th International Conference on Software Engineering and Formal Methods (SEFM'15) (LNCS)*, R. Calinescu and B. Rumpe (Eds.), Vol. 9276. Springer, 344–359. https://doi.org/10.1007/978-3-319-22969-0_24

[6] Maurice H. ter Beek, Ferruccio Damiani, Stefania Gnesi, Franco Mazzanti, and Luca Paolini. 2019. On the Expressiveness of Modal Transition Systems with Variability Constraints. *Sci. Comput. Program.* 169 (2019), 1–17. https://doi.org/10.1016/j.scico.2018.09.006

[7] Maurice H. ter Beek, Ferruccio Damiani, Michael Lienhardt, Franco Mazzanti, and Luca Paolini. 2019. Supplementary material for: "Static Analysis of Featured Transition Systems". https://doi.org/10.5281/zenodo.2616646

[8] Maurice H. ter Beek and Erik P. de Vink. 2014. Towards Modular Verification of Software Product Lines with mCRL2. In *Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'14) (LNCS)*, Tiziana Margaria and Bernhard Steffen (Eds.), Vol. 8802. Springer, 368–385. https://doi.org/10.1007/978-3-662-45234-9_26

[9] Maurice H. ter Beek and Erik P. de Vink. 2014. Using mCRL2 for the Analysis of Software Product Lines. In *Proceedings of the 2nd FME Workshop on Formal Methods in Software Engineering (FormaliSE'14).* IEEE, 31–37. https://doi.org/10.1145/2593489.2593493

[10] Maurice H. ter Beek, Erik P. de Vink, and Tim A. C. Willemse. 2017. Family-Based Model Checking with mCRL2. In *Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering (FASE'17) (LNCS)*, M. Huisman and J. Rubin (Eds.), Vol. 10202. Springer, 387–405. https://doi.org/10.1007/978-3-662-54494-5_23

[11] Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, and Franco Mazzanti. 2015. Using FMC for Family-based Analysis of Software Product Lines. In *Proceedings of the 19th International Software Product Line Conference (SPLC'15).* ACM, 432–439. https://doi.org/10.1145/2791060.2791118

[12] Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, and Franco Mazzanti. 2016. Modelling and analysing variability in product families: Model checking of modal transition systems with variability constraints. *J. Log. Algebr. Meth. Program.* 85, 2 (2016), 287–315. https://doi.org/10.1016/j.jlamp.2015.11.006

[13] Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, and Franco Mazzanti. 2019. States and Events in KandISTI: A Retrospective. In *Models, Mindsets, Meta: The What, the How, and the Why Not?*, T. Margaria, S. Graf, and K. G. Larsen (Eds.). LNCS, Vol. 11200. Springer, 110–128. https://doi.org/10.1007/978-3-030-22348-9_9

[14] Maurice H. ter Beek, Stefania Gnesi, and Franco Mazzanti. 2015. From EU Projects to a Family of Model Checkers. In *Software, Services and Systems*, R. De Nicola and R. Hennicker (Eds.). LNCS, Vol. 8950. Springer, 312–328. https://doi.org/10.1007/978-3-319-15545-6_20

[15] Maurice H. ter Beek, Axel Legay, Alberto Lluch Lafuente, and Andrea Vandin. 2015. Quantitative Analysis of Probabilistic Models of Software Product Lines with Statistical Model Checking. *Electron. Proc. Theor. Comput. Sci.* 182 (2015), 56–70. https://doi.org/10.4204/EPTCS.182.5

[16] Maurice H. ter Beek, Alberto Lluch Lafuente, and Marinella Petrocchi. 2013. 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. ACM, 10–17. https://doi.org/10.1145/2499777.2500722

[17] Maurice H. ter Beek and Franco Mazzanti. 2014. VMC: Recent Advances and Challenges Ahead. In *Proceedings of the 18th International Software Product Line Conference (SPLC'14)*, Vol. 2. ACM, 70–77. https://doi.org/10.1145/2647908.2655969

[18] Maurice H. ter Beek, Franco Mazzanti, and Aldi Sulova. 2012. VMC: A Tool for Product Variability Analysis. In *Proceedings of the 18th International Symposium on Formal Methods (FM'12) (LNCS)*, D. Giannakopoulou and D. Méry (Eds.), Vol. 7436. Springer, 450–454. https://doi.org/10.1007/978-3-642-32759-9_36

[19] Maurice H. ter Beek, Michel A. Reniers, and Erik P. de Vink. 2016. Supervisory Controller Synthesis for Product Lines Using CIF 3. In *Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques (ISoLA'16) (LNCS)*, T. Margaria and B. Steffen (Eds.), Vol. 9952. Springer, 856–873. https://doi.org/10.1007/978-3-319-47166-2_59

[20] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: a Literature Review. *Inf. Syst.* 35, 6 (2010), 615–636. https://doi.org/10.1016/j.is.2010.01.001

[21] Harsh Beohar, Mahsa Varshosaz, and Mohammad Reza Mousavi. 2016. Basic behavioral models for software product lines: Expressiveness and testing preorders. *Sci. Comput. Program.* 123 (2016), 42–60. https://doi.org/10.1016/j.scico.2015.06.005

[22] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. 2015. νZ – An Optimizing SMT Solver. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15) (LNCS)*, C. Baier and C. Tinelli (Eds.), Vol. 9035. Springer, 194–199. https://doi.org/10.1007/978-3-662-46681-0_14

[23] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. 2013. 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).* ACM, 355–364. https://doi.org/10.1145/2491956.2491976

[24] Brian Chess and Jacob West. 2007. *Secure Programming with Static Analysis.* Addison-Wesley.

[25] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. 1986. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. Program. Lang. Sys.* 8, 2 (1986), 244–263. https://doi.org/10.1145/5397.5399

[26] Andreas Classen. 2010. *Modelling with FTS: a Collection of Illustrative Examples.* Technical Report P-CS-TR SPLMC-00000001. University of Namur.

[27] Andreas Classen. 2011. *Modelling and Model Checking Variability-Intensive Systems.* Ph.D. Dissertation. University of Namur.

[28] Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. 2012. Model checking software product lines with SNIP. *Int. J. Softw. Tools Technol. Transf.* 14, 5 (2012), 589–612. https://doi.org/10.1007/s10009-012-0234-1

[29] Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. 2014. Formal semantics, modular specification, and symbolic verification of product-line behaviour. *Sci. Comput. Program.* 80, B (2014), 416–439. https://doi.org/10.1016/j.scico.2013.09.019

[30] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-François Raskin. 2013. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Trans. Softw. Eng.* 39, 8 (2013), 1069–1089. https://doi.org/10.1109/TSE.2012.86

[31] Andreas Classen, Pierre Heymans, Pierre-Yves Schobbens, and Axel Legay. 2011. Symbolic model checking of software product lines. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11).* ACM, 321–330. https://doi.org/10.1145/1985793.1985838

[32] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. 2010. 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).* ACM, 335–344. https://doi.org/10.1145/1806799.1806850

[33] Stephen A. Cook. 1971. The Complexity of Theorem-Proving Procedures. In *Proceedings of the 3rd Annual Symposium on Theory of Computing (STOC'71).* ACM, 151–158. https://doi.org/10.1145/800157.805047

[34] Maxime Cordy, Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. 2013. ProVeLines: A Product Line of Verifiers for Software Product Lines. In *Proceedings of the 17th International Software Product Line Conference (SPLC'13)*, Vol. 2. ACM, 141–146. https://doi.org/10.1145/2499777.2499781

[35] Maxime Cordy, Andreas Classen, Gilles Perrouin, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. 2012. Simulation-Based Abstractions for Software Product-Line Model Checking. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12).* IEEE, 672–682. https://doi.org/10.1109/ICSE.2012.6227150

[36] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. 2013. 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).* IEEE, 472–481. https://doi.org/10.1109/ICSE.2013.6606593

[37] Sjoerd Cranen, Jan Friso Groote, Jeroen J. A. Keiren, Frank P. M. Stappers, Erik P. de Vink, Wieger Wesselink, and Tim A. C. Willemse. 2013. An Overview of the mCRL2 Toolset and Its Recent Advances. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*

(TACAS'13) (LNCS), N. Piterman and S. A. Smolka (Eds.), Vol. 7795. Springer, 199–213. https://doi.org/10.1007/978-3-642-36742-7_15

[38] Ferruccio Damiani, Michael Lienhardt, and Luca Paolini. 2017. A Formal Model for Multi SPLs. In *Proceedings of the 7th International Conference on Fundamentals of Software Engineering (FSEN'17) (LNCS)*, M. Dastani and M. Sirjani (Eds.), Vol. 10522. Springer, 67–83. https://doi.org/10.1007/978-3-319-68972-2_5

[39] Ferruccio Damiani, Michael Lienhardt, and Luca Paolini. 2019. A formal model for Multi Software Product Lines. *Sci. Comput. Program.* 172 (2019), 203–231. https://doi.org/10.1016/j.scico.2018.11.005

[40] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08) (LNCS)*, C. R. Ramakrishnan and J. Rehof (Eds.), Vol. 4963. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[41] Rocco De Nicola and Frits W. Vaandrager. 1990. Action versus State based Logics for Transition Systems. In *Semantics of Systems of Concurrent Processes: Proceedings of the LITP Spring School on Theoretical Computer Science (LNCS)*, I. Guessarian (Ed.), Vol. 469. Springer, 407–419. https://doi.org/10.1007/3-540-53479-2_17

[42] Xavier Devroey, Gilles Perrouin, Axel Legay, Maxime Cordy, Pierre-Yves Schobbens, and Patrick Heymans. 2014. Coverage Criteria for Behavioural Testing of Software Product Lines. In *Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'14) (LNCS)*, T. Margaria and B. Steffen (Eds.), Vol. 8802. Springer, 336–350. https://doi.org/10.1007/978-3-662-45234-9_24

[43] Xavier Devroey, Gilles Perrouin, Mike Papadakis, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. 2016. Featured Model-based Mutation Analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. ACM, 655–666. https://doi.org/10.1145/2884781.2884821

[44] Aleksandar S. Dimovski. 2018. Abstract Family-Based Model Checking Using Modal Featured Transition Systems: Preservation of CTL*. In *Proceedings of the 21st International Conference on Fundamental Approaches to Software Engineering (FASE'18) (LNCS)*, A. Russo and A. Schürr (Eds.), Vol. 10802. Springer, 301–318. https://doi.org/10.1007/978-3-319-89363-1_17

[45] Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wąsowski. 2017. Efficient family-based model checking via variability abstractions. *Int. J. Softw. Tools Technol. Transf.* 5, 19 (2017), 585–603. https://doi.org/10.1007/s10009-016-0425-2

[46] Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wąsowski. 2015. Family-Based Model Checking Without a Family-Based Model Checker. In *Proceedings of the 22nd International SPIN Symposium on Model Checking of Software (SPIN'15) (LNCS)*, B. Fischer and J. Geldenhuys (Eds.), Vol. 9232. Springer, 282–299. https://doi.org/10.1007/978-3-319-23404-5_18

[47] Aleksandar S. Dimovski and Andrzej Wąsowski. 2017. Variability-Specific Abstraction Refinement for Family-Based Model Checking. In *Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering (FASE'17) (LNCS)*, M. Huisman and J. Rubin (Eds.), Vol. 10202. Springer, 406–423. https://doi.org/10.1007/978-3-662-54494-5_24

[48] Marijn Heule, Matti Järvisalo, and Martin Suda. [n.d.]. The international SAT Competitions web page. https://www.satcompetition.org/. Accessed: 2019-03-22.

[49] Gerald Holl, Paul Grünbacher, and Rick Rabiser. 2012. A systematic review and an expert survey on capabilities supporting multi product lines. *Inf. Softw. Technol.* 54, 8 (2012), 828–852. https://doi.org/10.1016/j.infsof.2012.02.002

[50] Frank Hutter, Marius Lindauer, Adrian Balint, Sam Bayless, Holger Hoos, and Kevin Leyton-Brown. 2017. The Configurable SAT Solver Challenge (CSSC). *Artifi. Intell.* 243 (2017), 1–25. https://doi.org/10.1016/j.artint.2016.09.006

[51] Christian Kästner and Sven Apel. 2008. Type-checking Software Product Lines – A Formal Approach. In *Proceedings of the 23rd International Conference on Automated Software Engineering (ASE'08)*. IEEE, 258–267. https://doi.org/10.1109/ASE.2008.36

[52] Chang Hwan Peter Kim, Don S. Batory, and Sarfraz Khurshid. 2011. Reducing Combinatorics in Testing Product Lines. In *Proceedings of the 10th International Conference on Aspect-Oriented Software Development (AOSD'11)*. ACM, 57–68. https://doi.org/10.1145/1960275.1960284

[53] Jan Křetínský. 2017. 30 Years of Modal Transition Systems: Survey of Extensions and Analysis. In *Models, Algorithms, Logics and Tools*, L. Aceto, G. Bacci, G. Bacci, A. Ingólfsdóttir, A. Legay, and R. Mardare (Eds.). LNCS, Vol. 10460. Springer, 36–74. https://doi.org/10.1007/978-3-319-63121-9_3

[54] Kim Guldstrand Larsen and Bent Thomsen. 1988. A Modal Process Logic. In *Proceedings of the 3rd Symposium on Logic in Computer Science (LICS'88)*. IEEE, 203–210. https://doi.org/10.1109/LICS.1988.5119

[55] Michael Lienhardt, Ferruccio Damiani, Simone Donetti, and Luca Paolini. 2018. Multi Software Product Lines in the Wild. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'18)*. ACM, 89–96. https://doi.org/10.1145/3168365.3170425

[56] Zohar Manna and Amir Pnueli. 1995. *Temporal Verification of Reactive Systems: Safety*. Springer. https://doi.org/10.1007/978-1-4612-4222-2

[57] Marcílio Mendonça, Andrzej Wąsowski, and Krzysztof Czarnecki. 2009. SAT-based Analysis of Feature Models is Easy. In *Proceedings of the 13th International Software Product Line Conference (SPLC'09)*. ACM, 231–240.

[58] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 2005. *Principles of Program Analysis*. Springer. https://doi.org/10.1007/978-3-662-03811-6

[59] Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. 2018. Variability-Aware Static Analysis at Scale: An Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 27, 4 (2018), 18:1–18:33. https://doi.org/10.1145/3280986

[60] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1 (2014), 6:1–6:45. https://doi.org/10.1145/2580950

[61] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Sci. Comput. Program.* 79 (2014), 70–85. https://doi.org/10.1016/j.scico.2012.06.002

[62] Mahsa Varshosaz, Harsh Beohar, and Mohammad Reza Mousavi. 2018. Basic behavioral models for software product lines: Revisited. *Sci. Comput. Program.* 168 (2018), 171–185. https://doi.org/10.1016/j.scico.2018.09.001