Model Checking Lots of Systems

Efficient Verification of Temporal Properties in Software Product Lines

Andreas Classen,[*]
Patrick Heymans,
Pierre-Yves Schobbens
University of Namur, Belgium
{acs,phe,pys}
@info.fundp.ac.be

Axel Legay
IRISA/INRIA Rennes, France
axel.legay@irisa.fr

Jean-François Raskin
Université Libre de Bruxelles,
Belgium
jraskin@ulb.ac.be

ABSTRACT

In product line engineering, systems are developed in *families* and differences between family members are expressed in terms of *features*. Formal modelling and verification is an important issue in this context as more and more critical systems are developed this way. Since the number of systems in a family can be exponential in the number of features, two major challenges are the scalable modelling and the efficient verification of system behaviour. Currently, the few attempts to address them fail to recognise the importance of features as a unit of difference, or do not offer means for automated verification.

In this paper, we tackle those challenges at a fundamental level. We first extend transition systems with features in order to describe the combined behaviour of an entire system family. We then define and implement a model checking technique that allows to verify such transition systems against temporal properties. An empirical evaluation shows substantial gains over classical approaches.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods, Model checking*

General Terms

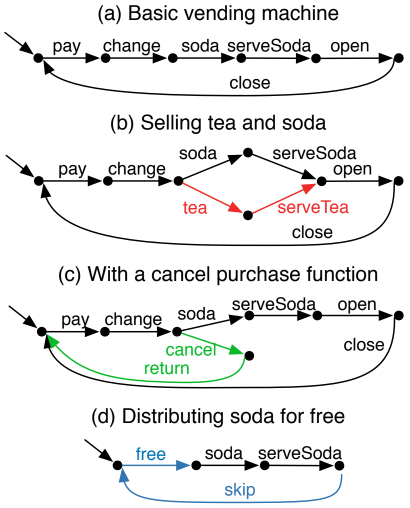
Algorithms, Reliability, Theory, Verification

Keywords

Software Product Lines, Features, Specification

1. INTRODUCTION

A software product line (SPL) is traditionally defined as *"a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way"* [10]. Software product line engineering (SPLE) promotes reuse throughout the software lifecycle in order to benefit from economies of scale when developing several (usually many) similar systems. SPLE proved beneficial to the development of embedded and critical systems [14], which makes formal modelling and verification in SPLE all the more important.


(a) Basic vending machine

(b) Selling tea and soda

(c) With a cancel purchase function

(d) Distributing soda for free

Figure 1: Several variants of a vending machine.

The differences between the systems of an SPL (i.e. its *variability*) are typically expressed in terms of *features*. In SPLE, features are first-class abstractions that shape the reasoning of the engineers and other stakeholders [8]. A set of features can be seen as the specification of a *product*, i.e. a particular *member* of the product line. Feature diagrams (FDs) [21, 31] are commonly used to model the variability of the SPL. An FD expresses the set of valid products, and since products are combinations of features, there might be an exponential number of them. For this reason, it is unrealistic to specify or verify the behaviour of each product individually. We illustrate these points with an example.

1.1 Motivating example

Throughout this paper, we use a beverage vending machine (inspired from [17]) as a running example. In its basic version, the vending machine takes a coin, returns change,

[*]FNRS Research Fellow

ICSE '10, May 2-8 2010, Cape Town, South Africa

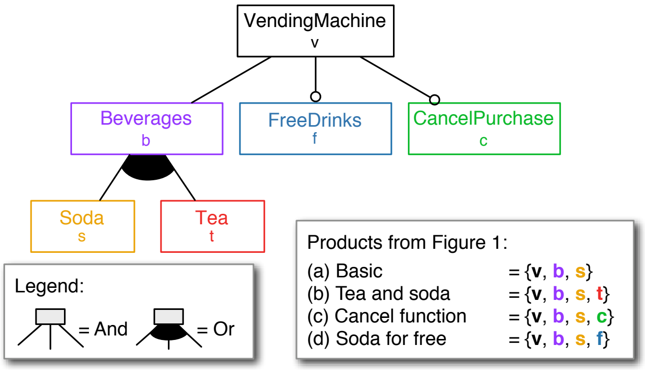


Figure 2: FD for the vending machines of Figure 1.

serves soda, and eventually opens a compartment so that the customer can take her soda, before it closes it again. This behaviour is modelled by the transition system (TS) shown in Figure 1(a). A number of variants of this basic machine can be considered, as for instance a machine that also sells tea, shown in Figure 1(b). A second variant lets the buyer cancel her purchase after entering a coin, see Figure 1(c). A third one offers free drinks and has no closing beverage compartment, see Figure 1(d).

By combining these variants, yet other vending machines can be obtained. In fact, these four products are part of a larger SPL, which in terms of *features* is modelled by the FD of Figure 2. Basically, this FD formally describes the set of vending machine variants. In this case, there are twelve of them. This means that a model of the behaviour of a small example such as this would already require twelve, largely identical, behavioural descriptions, four of which are shown in Figure 1. For realistic cases, this number is so high that it is outright impossible to verify, let alone model, each product individually.

1.2 Current challenges

The above example illustrates two challenges that model-based SPLE approaches need to address: (a) scalable modelling and (b) efficient verification of system behaviour. Current proposals are based on UML [34], modal transition systems [18, 17], modal I/O automata [23, 24], deontic logics [2] and CCS [20]. With the exception of [24], these proposals suffer from two main limitations, both of which are addressed in the present paper.

Firstly, their behavioural models often fail to recognise the importance of features as a unit of difference. This means that they capture different behaviours, but offer little to no means to relate products and their behavioural descriptions. They also cannot make use of information contained in variability (e.g., feature) models, such as the co-occurrence or mutual exclusion of two or more features. Secondly, none of the proposals provides concrete means for checking behavioural models against temporal properties. A more thorough discussion of related work is provided in Section 7.

1.3 Contribution

In this paper, we materialise the vision sketched in [9] by tackling the above challenges at a *foundational* level. Our first contribution is *featured transition systems* (FTS), a variant of transition systems designed to describe the combined behaviour of an entire system family. FTS has a parameterised semantics that allows to obtain the behaviour of each product of the SPL. The second contribution is a dedicated model checking technique supported by a proof-of-concept tool. The tool allows to verify LTL properties for all the products of an SPL at once, and pinpoints the products that violate (resp. satisfy) the properties. We applied the tool to a specification exemplar, the mine pump controller [22], in order to evaluate the approach empirically. On the 64-product SPL, our model checking algorithm was on average 3.5 (and up to 7) times faster than verifying all products separately with the classical algorithm.

The principal advantages of FTS over existing work are (i) the modelling of variability as a first-class citizen, (ii) the ability to reason about the whole product line, or subsets of it, (iii) the ability to model very detailed behavioural variations, (iv) a running and freely available model checking tool, and (v) the ability to take feature dependencies and incompatibilities into account.

The paper is structured as follows. Section 2 recalls the necessary background on FD and TS. FTS are introduced in Section 3 and the model checking approach is described in Section 4. The evaluation is reported in Section 5 and future work is discussed in Section 6. Section 7 surveys related work. Eventually, Section 8 concludes the paper.

2. BASE CONCEPTS

In this section, we recall basic concepts and definitions that will be used throughout the rest of the paper. We assume that the reader is familiar with automata theory and has basic knowledge of formal verification (otherwise, see [6, 4]).

We informally recall the definition of feature diagrams (FDs). Skipping the details, an FD is a tuple (N, r, DE) where N is a set of features, $r \in N$ is the root, and $DE \subseteq N \times N$ is the set of decomposition edges between features. The semantics of an FD d, noted $[\![d]\!]_{FD}$, is the set of valid products, i.e. a set of sets of features: $[\![d]\!]_{FD} \subseteq \mathcal{P}(N)$. As an example, the semantics of the vending machine FD from Figure 2 is as follows (using the short feature names):

$$\begin{gathered}\{\{v,b,t\},\{v,b,t,f\},\{v,b,t,c\},\{v,b,t,f,c\},\{v,b,s\},\\ \{v,b,s,f\},\{v,b,s,c\},\{v,b,s,f,c\},\{v,b,s,t\},\\ \{v,b,s,t,f\},\{v,b,s,t,c\},\{v,b,s,t,f,c\}\}.\end{gathered}$$

A complete formal definition of FDs can be found in [31].

In this paper, behaviour of individual products is represented with transition systems [4] (TS). A TS is a directed graph whose transitions are labelled with actions, and whose states are labelled with atomic propositions.[1] Formally, we have the following definition.

Definition 1 (Transition System). *A TS is a tuple $M = (S, Act, trans, I, AP, L)$ where*

- *S is a set of states,*
- *Act is a set of actions,*
- *$trans \subseteq S \times Act \times S$ is a set of transitions, with $(s_1, \alpha, s_2) \in trans$ sometimes noted $s_1 \xrightarrow{\alpha} s_2$,*
- *$I \subseteq S$ is a set of initial states,*
- *AP is a set of atomic propositions,*
- *$L : S \to 2^{AP}$ is a labelling function.*

[1] To avoid clutter, we omit atomic propositions in the figures.

An *execution* (also called behaviour) of M is a non-empty, infinite sequence $s_0\alpha_1 s_1\alpha_2\ldots$ with $s_0 \in I$ such that $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ for all $0 \le i < n$. The semantics of a TS, written $[\![t]\!]_{TS}$, is given by its set of executions.

In this paper, we mainly focus on two types of properties: (i) regular safety properties and (ii) ω-regular properties (among others, LTL properties). We follow the classical automata-based approach to model checking [33] which represents the complement of regular safety properties by a finite automaton (FA). Model checking of these properties is thus reduced to reachability in the synchronous product of this automaton and the TS. Similarly, the complement of ω-regular properties is classically represented by a Büchi automaton (BA). These properties can then be checked by repeated reachability in the synchronous product. The formal definition for FA and BA is given below. The language of an FA (resp. BA) consists of finite (resp. infinite) words and can be *empty* (accepts no word).

Definition 2. *An FA (resp.* BA*) is a tuple* $(Q, \Sigma, \delta, Q_0, F)$ *where* Q *is a set of states,* Σ *is the alphabet,* $\delta \subseteq Q \times \Sigma \times Q$ *the transition relation,* $Q_0 \subseteq Q$ *a set of initial states and* $F \subseteq Q$ *a set of accepting states. An FA accepts finite words that reach an accepting state, and a BA accepts infinite words that visit accepting states infinitely often.*

3. FEATURED TRANSITION SYSTEMS

In order to model the behaviour of each product in the SPL concisely, we draw upon existing approaches [12, 18, 20, 24] that create a single parameterised model to be instantiated differently for each product of the SPL. However, in contrast to most approaches, we explicitly relate behaviours to their originating features, and do this at the level of individual transitions.

3.1 Syntax

The syntax of FTS accounts for the fact that adding a feature to a system modifies the behaviour of this system. Consider the vending machine example. Figures 1(b,c and d) show the impact of adding features Tea, CancelPurchase and FreeDrinks, respectively, to a machine serving only soda. Tea adds two transitions (tea and serveTea); CancelPurchase also adds two transitions; and FreeDrinks replaces transitions pay and change by a single transition free as well as open/close by skip.

In order to describe the effects of several features on a system concisely, our approach models a system that contains all features, as well as annotations that indicate which transitions of the model correspond to which feature. To be able to express cases in which a feature *removes* (rather than adds) transitions, we use a priority relation over alternative transitions.[2] This leads us to define a *featured* TS as a TS in which each transition is labelled with a feature, and where a priority relation may be associated to transitions leaving the same state.

The FTS for the vending machine example is given in Figure 3. The feature label of a transition is shown next to its action label, separated by a slash. In addition, the transitions are coloured in the same way as the features in Figure 2. Intuitively, the FTS captures the impact of all features in a single diagram. Formally, FTS are defined as follows.

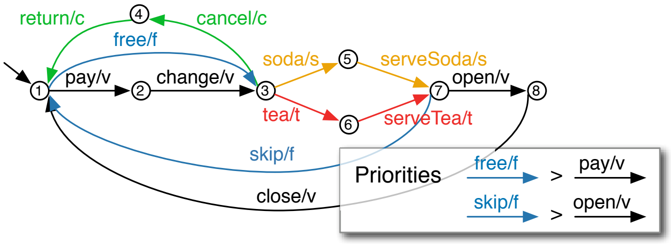


Figure 3: FTS of the vending machine.

Definition 3 (Abstract syntax of FTS). *An FTS* fts *is a tuple* $fts = (S, Act, trans, I, AP, L, d, \gamma, >)$ *where*

- $(S, Act, trans, I, AP, L)$ *is a TS,*
- $d = (N, r, DE)$ *is an FD,*
- $\gamma : trans \to N$ *is a total function, labelling transitions with features,*
- $> \subseteq trans \times trans$ *is a partial order, defining priorities between transitions.*

Transition priorities offer an intuitive way to model cases in which one feature overrides the behaviour of another. If a transition t labelled with feature f has priority over transition t' labelled with feature f', this means that products containing both f and f' only have transition t; they would have both transitions if there were no priority relation. The transition free of the FreeDrinks feature, for instance, has priority over pay (which belongs to the root feature, VendingMachine). The result is that pay will not appear in any product that contains the feature FreeDrinks, such as the one in Figure 1(d).[3]

A common modelling pattern is that the behaviour of a child feature (wrt. the FD) overrides the behaviour of its parents. Formally, we can use the decomposition relation DE of the FD (a directed graph), to induce the priority relation of the FTS.

Definition 4 (Priorities induced by FD). *A transition* $s \xrightarrow{\alpha} s_1$ *labeled with* f_1 *has priority over* $s \xrightarrow{\beta} s_2$ *labeled with* f_2*, written* $s \xrightarrow{\alpha} s_1 > s \xrightarrow{\beta} s_2$*, iff* f_2 *is an ancestor of* f_1 *in* DE*.*

The purpose of an FTS is to model the behaviour of the whole SPL. From the FTS, one can obtain the behaviour of one particular product through *projection*. Intuitively, the diagrams (a), (b), (c) and (d) of Figure 1 can be obtained by removing selected transitions from Figure 3.

Formally, in order to obtain the behaviour of a particular product, one *projects* the FTS on the corresponding set of features, say $p \in [\![d]\!]_{FD}$. This transformation is entirely syntactical and consists in removing (i) all transitions linked to features that are not in p, and (ii) all transitions that are overridden by higher priority transitions. The result of the projection is an ordinary TS.

[2] In essence, removing a transition corresponds to adding an alternative transition of higher priority.

[3] Since state 2 and transition change become unreachable, they are omitted from the diagram.

Definition 5 (Projection). *The projection of an FTS fts to a product $p \in [\![d]\!]_{FD}$, noted $fts_{|p}$, is the TS $t = (S$, Act, $trans'$, I, AP, $L)$ where*

$$\begin{aligned} trans' = \big\{(s_1,\alpha,s_2) | (s_1,\alpha,s_2) \in trans \;\wedge\; \gamma(s_1,\alpha,s_2) \in p \\ \wedge\; \nexists (s_1,\alpha',s_2') \in trans \bullet \gamma(s_1,\alpha',s_2') \in p \\ \wedge\; (s_1,\alpha',s_2') > (s_1,\alpha,s_2)\big\}. \end{aligned}$$

Note that the concept of *parallel composition* also exists for FTS. If the modelled SPL consists of several processes running in parallel, each process can be modelled as a separate FTS, all sharing the same underlying features and FD. The FTS of the system can then be obtained by composing these processes. For FTS, one can easily adapt the well-accepted handshake communication model, whereby the execution of parallel processes is synchronised on transitions with shared actions and otherwise interleaved.

3.2 Semantics

Each TS obtained through projection describes the behaviour (see Definition 1) of a particular product of the SPL. The semantics of an FTS is thus the union of the behaviours of the projections on all valid products.

Definition 6 (Semantics of an FTS).

$$[\![fts]\!]_{FTS} = \bigcup_{c \in [\![d]\!]_{FD}} [\![fts_{|c}]\!]_{TS}$$

An important observation is that, except for trivial cases, the FTS semantics we just defined is not equal to the usual TS semantics (as given by Definition 1). Formally, there exists an FTS fts for which $[\![fts]\!]_{FTS} \neq [\![TS(fts)]\!]_{TS}$, where $TS(fts)$ is the TS obtained by removing d, γ and $>$ from fts. The vending machine SPL is an example for such an FTS. Indeed, an execution e in which the vending machine would ask the first customer for a coin and offer a free drink to the next one would be part of $[\![TS(fts)]\!]_{TS}$, since in the TS, the choice between pay and free is non-deterministic. Yet, the execution does not correspond to any of the machines in the SPL: these should either always offer free drinks or always require payment, hence $e \notin [\![fts]\!]_{FTS}$. More generally, we have the following theorem.

Theorem 7 (FTS semantics vs. TS semantics).

$$\forall fts \bullet [\![fts]\!]_{FTS} \subseteq [\![TS(fts)]\!]_{TS}$$

This theorem illustrates that one cannot simply use classical model checking algorithms directly on an FTS to verify properties for the complete SPL. While this verification might be sound, it is not always complete: by ignoring priorities, it would find false positives. Definition 6 shows another problem that we have to face when model checking an FTS: the exponential blowup caused by considering all products of the SPL. This adds to the state-explosion problem that already exists in classical model checking.

These considerations justify the need for an FTS-specific model checking algorithm, but before we get there, we need a more operational definition of the FTS semantics.

3.3 Reachability in FTS

A model checker is meant to perform a search in the state space of the FTS and thus needs an execution model that is faithful to the FTS semantics. As we just showed, TS semantics does not apply to FTS. In addition, we want our model checker to indicate the products for which a property does (or does not) hold. To explore the state space of an FTS, a proper execution model thus needs to keep track of products and respect transition priorities.

Consequently, we define the FTS reachability relation R_0, to be constructed as the state space is explored. $R_0 \subseteq S \times \mathcal{P}(\mathcal{P}(N))$ is a set of couples (s, px) such that state s is reachable by the products in px. In particular, the initial states of the FTS are reachable for all products.

Definition 8. Initially reachable *states of an FTS are*

$$Init_0 \triangleq \big\{(s, [\![d]\!]_{FD}) \mid s \in I\big\}.$$

Given a state s reachable by products in px, a transition leaving s, say $t = s \xrightarrow{\alpha} s'$, can be fired for all products if its feature is part of all products in px and if there is no higher-priority transition $s \xrightarrow{\alpha'} s''$ that could also be fired. If these conditions hold, s' is reachable by the same products in px. In case the conditions do not hold, the transition cannot be fired for all products, that is, s' will only be reachable by a subset of px and we proceed as follows:

- If a higher-priority transition t' could equally be fired, then t can only be fired for (and thus s' is only reachable by) the products that do not contain the feature of t'.
- If the feature of t is part of some products in px only, it can only be fired for (and thus s' is only reachable by) these products.
- If none of the products contains the feature of t, it cannot be fired at all.

This is formalised in the following definition.

Definition 9. *The* successors *of a state $s \in S$ reachable by products in $px \in \mathcal{P}(\mathcal{P}(N))$ can be computed with the following operator*

$$\begin{aligned} &Post_0(s, px) \triangleq \\ &\Big\{(s', px') \mid s \xrightarrow{\alpha} s' \in trans \;\wedge \\ &\qquad px' = \big\{p \in px \mid \gamma(s \xrightarrow{\alpha} s') \in p \;\wedge \\ &\qquad\qquad \{\gamma(s \xrightarrow{\alpha'} s'') \mid s \xrightarrow{\alpha'} s'' > s \xrightarrow{\alpha} s'\} \cap p = \emptyset\big\}\Big\}. \end{aligned}$$

Let us illustrate this with the vending machine FTS of Figure 3. State 1 is an initial state, and thus reachable by all products. From there, the transition pay can only be fired by products containing the feature v (the label of pay), and not containing the feature f (the label of the higher-priority transition free). State 2 is thus reachable by these products only. From state 2, transition change can be fired for all products containing v, and so state 3 is reachable by the same products as state 2.

Recording the set of products would be too expensive. The set of reachable states will be of size $O(|S|.2^{2^{|N|}})$. We propose a more concise representation for a set of products: to state which features the products must have (required features, rf) and which they cannot have (excluded features, ef). This *symbolic* data structure is defined as follows.

Definition 10. *A triple $(s, rf, ef) \in S \times \mathcal{P}(N) \times \mathcal{P}(N)$ is a symbolic encoding of a tuple $(s, px) \in S \times \mathcal{P}(\mathcal{P}(N))$ such that $[\![(s, rf, ef)]\!] \triangleq (s, \{p \in [\![d]\!]_{FD} \mid rf \subseteq p \wedge ef \cap p = \emptyset\})$.*

The new, efficient reachability relation R is thus a set of triples (s, rf, ef), where the initially reachable states ($Init$) and the successors ($Post$) are defined as follows.

Definition 11. *In symbolic representation, the initially reachable states of an FTS are* $Init \triangleq \{(s, \emptyset, \emptyset) \mid s \in I\}$*; the successors of a state* $s \in S$ *reachable by products containing features* $rf \subseteq N$ *and not containing features* $ef \subseteq N$ *are*

$$\begin{aligned} & Post(s, rf, ef) \triangleq \\ & \quad \{(s', rf', ef') \mid \quad s \xrightarrow{\alpha} s' \\ & \qquad \wedge\ rf' = rf \cup \{\gamma(s \xrightarrow{\alpha} s')\} \\ & \qquad \wedge\ ef' = ef \cup \textstyle\bigcup_{s \xrightarrow{\alpha'} s'' > s \xrightarrow{\alpha} s'} \gamma(s \xrightarrow{\alpha'} s'') \\ & \qquad \wedge\ ef' \cap rf' = \emptyset\}. \end{aligned}$$

It can easily be shown that the symbolic successor function is equivalent to its explicit counterpart, that is:

Theorem 12. *For any* (s, rf, ef)*,*

$$Post_0([\![(s, rf, ef)]\!]) = [\![Post(s, rf, ef)]\!]$$

where $[\![.]\!]$ *is trivially extended to sets of triples.*

Using this representation, the size of the reachability relation will be $O(|S|.2^{|N|}.2^{|N|})$. Since triples with $rf \cap ef \neq \emptyset$ can be ignored, this shrinks to $O(|S|.3^{|N|})$, which is significantly smaller than what we had previously.

Its size can be further reduced by exploiting the following property: if a state s is known to be reachable by products in px, then it is also reachable by the products in any subset of px. Formally, if $(s, px) \in R_0$ and $(s, px') \in R_0$ with $px' \subseteq px$, it is sufficient to only keep (s, px) in R_0. More generally, it is sufficient to keep the maximal elements (an antichain) of the partial order induced by the subset relation $\subseteq$ over $\{px \mid (s, px) \in R_0\}$ for each state s. In terms of the symbolic representation that we are using, an equivalent partial order can be defined as follows.

Definition 13. *For a state* $s \in S$ *and a set* $R \subseteq s \times \mathcal{P}(N) \times \mathcal{P}(N)$*, the relation* $\sqsubseteq$ *is defined as a partial order over R:* $(s, rf, ef) \sqsubseteq (s, rf', ef') \triangleq (rf \supseteq rf') \wedge (ef \supseteq ef')$.

With this optimisation, testing whether a state s is reachable by products in px cannot be done by just checking whether $(s, px) \in R_0$. Indeed, if the state s is reachable by a greater set of products px' with $px \subseteq px'$, only (s, px') will be in R_0. One therefore has to check whether $\exists (s', px') \in R \bullet s = s' \wedge px \subseteq px'$. In the symbolic representation, for a state (s, rf, ef), this boils down to checking whether $\exists (s', rf', ef') \in R \bullet s = s' \wedge (s, rf, ef) \sqsubseteq (s', rf', ef')$.

4. MODEL CHECKING FTS

Our objective is to verify regular and ω-regular properties in such a way that (a) if a property is satisfied by the FTS, then it is also satisfied by every product of the SPL, and (b) if a property is violated, the algorithm reports a counterexample (a trace that violates the property) as well as the products of the SPL that violate the property.

This differs from classical model checking algorithms which, in case of a violation, just return the counterexample. In SPL model checking, information about the violating products is needed to help the engineer correct the model.

4.1 FTS model checking scenarios

We first need to define what it means for an FTS to be a model of a temporal property. As stated in the following definition, an FTS satisfies a temporal property if all its projections satisfy the property.

Definition 14 (Satisfaction in FTS). *An FTS fts satisfies a (regular, or* ω*-regular) property* ϕ*, iff*

$$\forall p \in [\![d]\!]_{FD} \bullet fts_{|p} \models \phi.$$

Extending the $\models$ *relation, we note this* $fts \models \phi$*.*

The FTS model checking problem can now be formalised.

Definition 15 (MC(fts, ϕ)). *Given a property* ϕ *and an FTS fts,* $MC(fts, \phi)$ *returns* true *iff* $fts \models \phi$*. If* $fts \not\models \phi$*, it returns* false*, a counterexample e, and a non-empty set of products* $px \subseteq [\![d]\!]_{FD}$ *such that* $\forall p \in px \bullet fts_{|p} \not\models \phi$ *with e as counterexample.*

The basic model checking scenario is analogous to classical model checking: just as the returned counterexample might be one out of many violating traces, the set of violating products is not necessarily complete. In case of a violation, it is therefore not possible to know whether there are products that *do* satisfy the property. This gives rise to an SPL-specific model checking problem: determine which products satisfy and which violate the property.

Definition 16 (ExtMC(fts, ϕ)). *Given a property* ϕ *and an FTS fts,* $ExtMC(fts, \phi)$ *returns* true *iff* $fts \models \phi$*. If* $fts \not\models \phi$*, it returns* false *and a set c of couples* (e, px) *where e is a counterexample and px a non-empty set of products such that* $\forall p \in px \bullet fts_{|p} \not\models \phi$*. Furthermore, it holds that*

$$\forall p \in [\![d]\!]_{FD} \bullet p \notin \bigcup_{(e,px) \in c} px \implies fts_{|p} \models \phi.$$

The last condition of the above definition states that the list of counterexamples has to be exhaustive, i.e. all products that are not mentioned satisfy the property. The procedure thus implicitly returns a set of violating and a set of satisfying products. A further variation of these two scenarios is useful for SPLE: limiting the verification to a subset of the products of the SPL. Basically, both scenarios would take $px \subseteq [\![d]\!]_{FD}$, the set of products to verify, as an additional parameter. From there on, the definitions are analogous.

4.2 Synchronous product

As stated in Section 2, we follow the approach of automata-based model checking [33], where regular and ω-regular properties are expressed by automata. In this case, model checking is equivalent to checking whether or not the *synchronous product* of the system with the automaton representing the *negation* of the property has an empty language.

The synchronous product of an FTS and an automaton is similar to that of a TS and an automaton [4]. That is, it uses the state labelling (the atomic propositions) of the FTS, and not the transition labels (as the parallel composition does). The difference from the standard definition is that it has to preserve feature labels and priorities of the original FTS.

Definition 17 (Synchronous product). *For an FTS* $fts = (S, Act, trans, I, AP, L, d, \gamma, >)$ *and an FA/BA* $a = (Q, \mathcal{P}(AP), \delta, Q_0, F)$*, the synchronous product is an FTS* $fts \otimes a = (S \times Q, Act, trans', I', AP', L', d, \gamma', >')$*, where*

- $AP' = Q$ *and* $L'(s,q) = q$, *i.e. the new FTS is labeled with the states of the FA/BA,*
- $(s,q) \overset{\alpha}{\rightarrow}' (t,p)$ *iff* $s \xrightarrow{\alpha} t \ \wedge\ q \xrightarrow{L(t)} p$,
- $I' = \{(s_0,q) \mid s_0 \in I \ \wedge\ \exists q_0 \in Q_0 \bullet (q_0, L(s_0), q) \in \delta\}$, *i.e. the initial states are those that can be reached from an initial state of the FA/BA,*
- $\gamma'\big((s,q) \overset{\alpha}{\rightarrow}' (t,p)\big) = \gamma(s \xrightarrow{\alpha} t)$,
- $(s,q) \overset{\alpha}{\rightarrow}' (t,p) >' (s,q) \overset{\alpha'}{\rightarrow}' (t',p')$ *iff* $s \xrightarrow{\alpha} t > s \xrightarrow{\alpha'} t'$.

Note that the synchronous product of an FTS fts and an automaton a is an FTS fts', not an automaton. Its language, though, can be defined in the same way as for FA/BA in Definition 2. The accepting states are the states that are labelled with an accepting state of a, $\{s \in S \times Q | L'(s) \in F\}$ and the words are the executions $e \in [\![fts']\!]_{FTS}$.

4.3 Model checking regular safety properties

To prove that an FTS fts satisfies a regular property ϕ, the latter is negated and transformed into an FA: $FA(\neg\phi)$. $FA(\neg\phi)$ is then composed with the FTS: $fts \otimes FA(\neg\phi)$ yielding a new FTS fts' which has to be proven empty. Conversely, to prove that fts *violates* ϕ is to prove that fts' has an accepting run. This boils down to checking whether an accepting state, a 'bad' state s with $L'(s) \in F_{FA(\neg\phi)}$, is reachable in fts'.

This is accomplished with a search in the reachability relation R, as discussed in Section 3.3. The easiest way to do this is by computing a fixpoint: the reachable states are the initially reachable states and those that can be reached from them, i.e. the least fixpoint of the successor operator [6].

Definition 18. *The symbolic reachability relation* $R \subseteq S \times \mathcal{P}(N) \times \mathcal{P}(N)$ *for an FTS is defined as*

$$R = \mu X \bullet Init \cup Post(X),$$

where $Post$ is extended to sets of triples as follows: $Post(x) = \bigcup_{(s,rf,ef)\in x} Post(s, rf, ef)$. The relation can be calculated following Tarski's fixpoint theorem by applying the successor relation until it stabilises, i.e.

$$R = Init \cup \bigcup_{i\geq 1} Post^i(Init).$$

Alternatively, one can check reachability directly with a depth-first search (DFS) in the FTS. The advantage of a DFS is that it is more natural to obtain a counterexample for a violated property. Our DFS algorithm is given in the procedure `IsReachable` (see right column of this page). The procedure takes five parameters so that it can be used for all model checking scenarios identified in Section 4.1:

(i) the FTS, that is: $fts \otimes FA(\neg\phi)$;

(ii) the set of accepting states of $FA(\neg\phi)$: $F_{FA(\neg\phi)}$;

(iii) a flag instructing the procedure to stop upon discovery of a reachable state: $true$ to compute $MC(fts,\phi)$ and $false$ to compute $ExtMC(fts,\phi)$;

(iv/v) and the set of products to be verified (using the symbolic representation from Definition 10) in case the property should be checked for a subset of the products only (as discussed in Section 4.1).

Input: An FTS $fts = (S, Act, trans, I, AP, L, d, \gamma, >)$, a set of accepting states $F \subseteq AP$, a flag *break* instructing to stop upon discovery of a bad state, a set of required (resp. excluded) features rf_0 and ef_0 to delimit the products to explore.
Output: *True* if a state s with $L(s) \in F$ was found and a set of quadruplets (state, set of required, set of excluded features, error trace) with the violations, otherwise $false$.

```
1  $R \leftarrow \{(s_0, rf_0, ef_0) \mid s_0 \in I\}$ ;          % reachable states
2  $Trace \leftarrow []$ ;                                        % current trace
3  $bad \leftarrow \emptyset$ ;                                   % set of bad states
4  while $I \neq \emptyset$ do
5      Take $s_0$ from $I$;
6      $I \leftarrow I \setminus \{s_0\}$;
7      $push((s_0, rf_0, ef_0), Trace)$;
8      while $Trace \neq []$ do
9          $(s, rf, ef) \leftarrow top(Trace)$;
10         if $L(s) \in F$ then
11             $bad \leftarrow bad \cup \{(s, rf, ef, Trace)\}$;
12             if $break$ then return $true$, $bad$
13         end
14         $unvisited \leftarrow \left\{ (s', rf', ef') \in Post(s, rf, ef) \;\middle|\; \nexists (s', rf'', ef'') \in R \bullet (s', rf', ef') \sqsubseteq (s', rf'', ef'') \right\}$;
15         if $unvisited = \emptyset$ then
16             $pop(Trace)$
17         else
18             Take $(s', rf', ef') \in unvisited$;
19             $R \leftarrow max_{\sqsubseteq}(R \cup \{(s', rf', ef')\})$;
20             $push((s', rf', ef'), Trace)$
21         end
22     end
23 end
24 return $(bad \neq \emptyset), bad$
```

Procedure `IsReachable`$(fts, F, break, rf_0, ef_0)$

With these parameters, the procedure checks whether there is a trace violating ϕ and thus returns $true$ iff $MC(fts,\phi)$ (resp. $ExtMC(fts,\phi)$) is false. The violating products are then returned in the compact symbolic representation.

The procedure basically computes the symbolic reachability relation defined in Section 3.3. It maintains the set of reachable states R, a stack T of triples (s, rf, ef) and a set of property violations *bad*. The initial states are always reachable for all products that are going to be verified, and R is initialised accordingly (line 1). The procedure iterates over the initial states (line 4) and performs a DFS for each of them (line 8). It first checks whether the current state is bad (line 10); if it is, the current trace (that is, the counterexample) and the products for which the bad state is reachable (rf, ef) are saved (line 11). If the *break* flag is set, the procedure will terminate here (line 12).

The procedure continues by calculating the set of unvisited successors of the current state (line 14). This calculation uses the $Post$ operator from Definition 11 to determine the successors, and filters out those that are already in R (with the antichain optimisation discussed in Section 3.3). If all successors were visited, the procedure backtracks (line 16). Otherwise the search proceeds with one of the successor states, which is added to R (line 19) again using the antichain optimisation detailed in Section 3.3 (intuitively, $max_{\sqsubseteq}$ removes the redundant triples from R).

In its current form, the procedure does not take the structural information of the FD into account. This means that a symbolic couple (rf, ef) might designate also some invalid products of the FD, but would be considered during the DFS regardless. The DFS would thus visit states that are not actually reachable. While this overhead could be deemed acceptable, it is a problem if the procedure finds an alleged bad state that is actually not a bad state because it does not belong to a valid product. In order to address this, one could easily add a validity check after line 14 so that only unvisited states valid wrt. the FD are kept, i.e. $\{(s, rf, ef) \mid \exists p \in [\![d]\!]_{FD} \bullet rf \subseteq p \wedge ef \cap p = \emptyset\}$. A straightforward and efficient implementation of this check can be done with a SAT solver [27].

4.4 Model checking ω-regular properties

Verification of an ω-regular property ϕ, e.g. expressed in LTL, is similar. The property is negated and transformed into a BA: $BA(\neg\phi)$. This BA composed with the FTS to be checked, yielding a new FTS: $fts' = fts \otimes BA(\neg\phi)$, has to be proven empty [33]. The difference wrt. the previous case is the Büchi acceptance condition which requires that an accepting (bad) state be visited infinitely often.

Algorithmically, this can again be done by a search in the reachability relation R. A bad state has to be found that is reachable from an initial state (as before) and reachable from itself (on a cycle). While this could be done by a fixed-point calculation [15], we propose a DFS-based approach for the reasons stated above. Due to space restrictions, we omit the details of the algorithm, called `IsPersistent` hereafter. The principal differences from the classical model checking algorithm are sufficiently illustrated in Section 4.3. `IsPersistent` performs a double DFS [11]: the outer DFS searches for a reachable bad state (identical to `IsReachable`) and once one is found, an inner DFS checks whether the state is on a cycle. The optimisations of Section 3.3 (symbolic representation of products, antichain for the reachability relation) are used for both the inner and the outer DFS.

The procedure takes the same parameters as `IsReachable` (except that $\neg\phi$ is translated into a BA), and has the same return values: $true$ if the property is violated, and $false$ if it is satisfied. Since the procedure checks persistence instead of mere reachability, its counterexample is an infinite trace: the prefix of the trace consists of the transitions that lead to the bad state (found in the outer DFS) while the remainder consists of the transitions leading back to itself (found in the inner DFS), which can be repeated indefinitely.

5. EVALUATION

5.1 Theoretical evaluation

The bottleneck of automata-based model checking is the construction of the BA that accepts the negation of the property ϕ to be checked. LTL model checking by automata-based techniques is therefore $O(|TS|.2^{|\phi|})$. Furthermore, the LTL model checking problem is PSPACE-Complete [30].

LTL model checking of an FTS with n features against a property ϕ with our algorithm is $O(|FTS|.2^{|\phi|+nlog9})$. That is, the complexity of `IsPersistent`, $O(|FTS|.9^n)$, multiplied by the factor $2^{|\phi|}$ for the LTL to BA translation. The complexity of `IsPersistent` is the same as for `IsReachable` and obtained as follows. The loop at line 8 will explore the reachability relation, it is thus $O(|FTS|.3^n)$. In that loop, line 14 is the most costly step: $O(3^n)$ since it has to go through the fragment of the reachability relation belonging to a single state (although we believe that an efficient implementation could reduce that to $O(2^n)$). The complexity of `IsPersistent` follows from that, since it basically executes two such DFS (double DFS optimisation [11] can be used for FTS, too). Just as in LTL model checking, the procedure is thus linear in the size of the state space. The FTS LTL model checking problem is also PSPACE-Complete, by an argument similar to [30].

5.2 Empirical evaluation

We implemented the FTS model checking technique described in the previous section in Haskell, a functional programming language. The tool[4] comes in the form of a library that can be loaded into a Haskell interpreter to be accessed through a command line interface, or compiled to perform verifications in batch mode. The advantage of using Haskell is its pervasive use of lazy evaluation and the natural translation of mathematical formulae into program code. The tool interfaces with *ltl2ba*,[5] based on [19], to automate the translation from LTL to BA, and uses Graphviz[6] to render FTS graphically.

In order to evaluate our approach, we conducted a study of examples found in the literature (see [7]). Here, we report on the analysis of the mine pump controller exemplar [22]. The purpose of the system is to keep a mine shaft clear of water while avoiding the danger of a methane related explosion. It consists of a water pump, a sensor measuring the water level and a sensor measuring the abundance of methane in the mine. The system is supposed to activate the pump once the water level reaches a preset threshold, but only if the methane is below a critical limit.

The system, as designed in [22], is composed of a base system, base, and three high level features: c, a command interface that (de)activates water regulation; m, a methane alarm interface; and l, the water regulator itself. The system and its environment are modelled with five separate FTS (detailed in [7]): the main FTS represents the control structure of the program, a second FTS models the changes to the system state, and three other FTS model the state of the environment: the water level, the methane level and the state of the pump. The system FTS is the parallel composition of these FTS. It has 457 states and 1306 transitions, and the priority relation is empty. We introduce variability by modelling c and a as optional features. With only these high level features, the SPL has four products. Counting all products explicitly, the system has 1828 states and 4612 transitions. In a second step, we introduced more variability by further decomposing the high level features. Transitions in the main FTS were just labelled differently; no transitions were added or removed. This second SPL has nine features and 64 products. The explicit count here is 29760 states and 69856 transitions.

We used our prototype to prove properties such as those

[4] Download at `www.info.fundp.ac.be/~acs/fts`

[5] `www.lsv.ens-cachan.fr/~gastin/ltl2ba`

[6] `www.graphviz.org`

Table 1: Benchmark results for exhaustive counter example search $ExtMC(\mathbf{FTS}, \phi)$.

		4 features, 4 products			9 features, 64 products		
Formula ϕ		**Cur.**	**Our**	**Diff.**	**Cur.**	**Our**	**Diff.**
(1.1) $\Box\Diamond(start \wedge \bigcirc msg \wedge (methane \Rightarrow palarm)) \Rightarrow (\Box\Diamond(methane \Rightarrow \Diamond pumpoff)$	✓	9.389 s	5.563 s	1.69	57.706 s	8.162 s	7.07
(1.2) $\neg\Box\Diamond(start \wedge \bigcirc msg \wedge (methane \Rightarrow \Diamond palarm))$	✗	25.741 s	37.663 s	0.68	138.970 s	102.716 s	1.35
(1.3) $\Box\Diamond(start \wedge \bigcirc msg) \Rightarrow \Box(pumpon \Rightarrow \Diamond running)$	✓	5.084 s	4.308 s	1.18	13.716 s	5.317 s	2.58
(1.4) $\Box\Diamond(msg \wedge \bigcirc level) \Rightarrow \Box\Diamond(lowwater \Rightarrow \Diamond pumpoff)$	✓	4.970 s	4.156 s	1.20	16.450 s	4.926 s	3.34
(1.5) $\Box\Diamond(msg \wedge \bigcirc level \wedge ready) \Rightarrow \Box((highwater \wedge \Box !methane) \Rightarrow \Diamond pumpon)$	✓	5.172 s	4.462 s	1.16	14.981 s	5.033 s	2.98
(1.6) $\Box\Diamond(msg \wedge \bigcirc level \wedge ready) \Rightarrow \Box((highwater \wedge !methane) \Rightarrow \Diamond pumpon)$	✗	5.437 s	4.405 s	1.23	17.741 s	4.914 s	3.61

identified in [1] for both SPLs, comparing the performance of the classical model checking algorithm (also implemented in the tool) to our method. All benchmarks were run on a MacBook Pro with a 2,4 GHz Core 2 Duo processor and 4 Gb of RAM; the library was compiled using GHC.[7] Each benchmark comprises the construction of the BA, the computation of the parallel composition, of the synchronous product and of an exhaustive counterexample search. The reported runtime is the average of three executions. The results are listed in Table 1, where ✓(resp. ✗) means property satisfied (resp. violated), 'Cur.' is the time needed to verify all products separately with the classical algorithm, 'Our' is the runtime of our algorithm, and 'Diff.' the speedup factor.

Property (1.1), for instance, requires that *"the pump shall eventually be off when the methane level is critical."* The formal property needs a fairness assumption: $\Box\Diamond start \wedge \bigcirc msg$ forces the system to progress, preventing traces that only contain environment transitions. And $\Box\Diamond methane \Rightarrow palarm$ is the assumption that in case of critical methane, the system will eventually be notified with an alarm message. Property (1.2) is used to check that the assumption actually holds for some products. This assumption indeed holds for products that have features `base` and `m`, which means that property (1.1) also holds for these products.

These results show that even with a very low number of products, our approach achieves an average 20% improvement over classical algorithms. Exceptions appear in cases where the number of counterexamples is high, such as for property (1.2). However, the gains over the classical approach increase dramatically with the number of products. With 64 products (which still isn't large), we are on average 3.5 times faster, up to 7 times faster for property (1.1). Furthermore, the results show that our approach, as opposed to the classical one, scales with the number of products. It is noteworthy that both implementations are rather naive and do not make use of known optimisation techniques. Their primary purpose is to benchmark the speedup resulting from our algorithms. These initial results are encouraging and motivate us to pursue further evaluations and optimisations. Other future work is described in the next section.

6. FUTURE WORK

A natural generalisation of FTS would be to label each transition with a Boolean expression over the set of features, in the spirit of [12, 13]. Definition 5 (projection) would have to be adapted so that a transition becomes part of the projected TS only if the interpretation of the product satisfies its Boolean expression. This would provide greater flexibility when modelling. For instance, one could express situations in which a transition belongs to several features. Conceptually, such an extension is straightforward. The reachability relation will consist of couples (s, b) where s is a state and b is a Boolean expression characterising the products in which s is reachable. There are no fundamental changes required in the algorithms, although an efficient implementation would have to use BDDs or a SAT solver.

This paper assumes that the state-space of the SPL is given in the form of a single (or multiple parallel) FTS. While this is not a problem for illustrative examples and small cases, one may question the applicability to industrial systems. This is why we are planning to define translations from high-level modelling languages, such as Statecharts and Promela, to FTS. The intention of the present work is to lay the foundations of formal verification of SPL, we thus do not expect an engineer to model directly in FTS.

Even if the engineers use a high-level modelling language, it is likely that the full specification of an SPL cannot be created as a single model. We are thus also exploring merging techniques, which create an FTS of the SPL based on the TS fragments of high-level features [9].

7. RELATED WORK

SPL-specific approaches. Before we examine each related approach in detail, we note that they can be broadly categorised along two lines. On the one hand, there are approaches that provide a modelling language *with* verification mechanisms [18, 20, 24], as we do, and others that just provide the modelling language. Among the latter, there are a number of formal approaches that do not provide mechanisms for the verification of temporal properties [23, 17, 16, 2, 32] as well as UML-based approaches [34, 12, 13, 26] where family models can be used to syntactically derive the model of a specific product, but not be verified against temporal properties. One the other hand, one can distinguish between approaches that consider *variability as a first-class citizen* [12, 13, 24, 26] and those that express *variability as part of the behavioural model* [34, 18, 23, 17, 20, 16, 32, 2]. In [3], Bachmann *et al.* propose orthogonal variability modelling (OVM), a modelling paradigm that consists in documenting variability as first-class citizen in a separate model, which is related to the other (e.g., data or behavioural) models, called *base models*. We believe that OVM has a number of important benefits, the most important being a clear separation of concerns. Approaches that represent variability as part of the behavioural model are problematic for several

[7] `www.haskell.org/ghc`

reasons. Firstly, they bury variability information inside a behavioural model although variability crosscuts all kinds of models, not only behaviour. Secondly, when variability information is scattered across base models, the variability in these models has to be kept in sync. For instance, as the product line evolves, optional functionality might become mandatory, requiring similar changes in all base models. Thirdly, variable artefacts in these approaches do not have an identity other than what is provided by the modelling language, making it hard to explicitly capture the notion of a *product* as a set of features or decisions.

Ziadi *et al.* [34] propose a UML profile for variability with stereotypes for optionality, alternatives and refinement (called 'virtuality'). This profile can be used to model product line behaviour with UML sequence diagrams. The approach does not provide verification mechanisms nor does it use a first-class variability approach.

Czarnecki *et al.* [12, 13] propose a pruning-based approach to UML modelling of SPL that separates variability from the base models. They propose to annotate model fragments with 'presence conditions', i.e. Boolean expressions over features that define to which products a fragment belongs. The authors do not deal with semantic issues and only verify syntactical correctness of possible projections.

Larsen *et al.* propose modal I/O automata as a way to model configurable components and provide a formal notion of compatibility between components [23]. Their notion of variability is limited to that of variable component interfaces and the approach does not deal with the problem of specifying variable behaviour in order to verify temporal properties.

Modal transition systems (MTS) were first proposed by Fischbein *et al.* [18] to model SPL behaviour. Transitions in an MTS are mandatory or optional. An MTS thus specifies a family of behaviours since optional transitions may or may not be fired when executing. Similarly to our approach, a single MTS model check allows to verify all possible products at once. Yet, MTS lack the notion of feature and priority between transitions. Intuitively, they execute without a memory of the decisions taken.

Fantechi and Gnesi [17, 16] extended this approach by introducing explicit variability operators into MTS, similar to the way Ziadi *et al.* did for UML. In addition to optionality, they allow to specify cases in which $i..j$ outgoing transitions may be taken. This proposal does not overcome the inherent MTS limitation that individual variation points are unrelated. Asirelli *et al.* show how an MTS can be completely characterised with deontic logic formulae [2], which means that deontic logic is as expressive as MTS and could be used as a specification language as well.

Gruler *et al.* propose PL-CCS, a variant of CCS extended with a product line variant operator that allows to model an alternative choice between two processes [20]. The goal of their verification procedure is similar to ours: verify all systems of the product line at once. However, their model checking procedure is only sketched and, as far as we know, no implementation is available. Also, their properties have to be expressed with multi-valued modal μ-calculus, whereas we use the more widely accepted LTL. Moreover, we do not introduce a new operator, making it easier to adapt existing tools for our approach. Finally, we believe that modelling variability with the alternative choice operator can result in verbose descriptions since common parts of the alternatives have to be duplicated; see [7] for a side-by-side comparison of FTS and PL-CCS using the example from Gruler *et al.*

Model checking of CTL properties in SPLE is addressed by Lauenroth *et al.* with an approach based on automata labeled with features [24]. While similar to FTS, their modelling language does not support priorities between features, and uses a non-standard definition of the parallel composition (which adds transitions that were not in the original automata). Their algorithms are of higher computational complexity: $O(|\phi|.|A|!) = O(|\phi|.|A|^{|A|})$, where $|A|$ is the size of the (generally huge) state space.

For safety analysis in SPLE, Liu *et al.* [26] use Statecharts to model (parts of) SPL components. As in [12, 13], instances can be derived syntactically by pruning. Each possible instance is manually run against a bad scenario to check whether or not it may occur. There is no support for verification of arbitrary temporal properties.

Other approaches. In addition to the above, there is a body of related research in the field of feature interaction detection [5]. The purpose of these approaches is to detect and manage incompatibilities, called *harmful interactions*, between features (mostly in telecommunication systems). Feature interaction research lacks the product line perspective: their techniques generally focus on pair-wise checks and do not deal with the problem of an exponential number of possible feature combinations. Also, their purpose is generally not the verification of behavioural models against arbitrary properties. An exception is the approach by Plath and Ryan [29], which allows verification of arbitrary properties, but is restricted to pair-wise checks.

A compositional approach for CTL model checking is proposed by Li *et al.* [25]. A feature automaton can be attached to two precisely defined interface states of the base system. The advantage of this approach is that each feature can be verified in isolation. The disadvantage is lost expressiveness: features can only add sequentially at the interface and not at several places at the same time. Furthermore, features cannot remove transitions or states.

Morin *et al.* propose a method to check for inconsistencies between features in adaptive systems [28]. Instead of verifying all possible combinations at design time, they verify a feature combination when it is activated at runtime, which is prevented in case of an inconsistency. However, their verification only covers structural properties of the system.

In the context of workflow modelling, van der Aalst *et al.* propose workflow templates that contain variation points [32]. The authors propose a technique for configuring workflow models incrementally, continuously verifying that they are deadlock free. Their approach does not generalise to checking arbitrary user-defined properties.

8. CONCLUSION

This paper lays the foundations for scalable modelling and efficient verification of software product lines. We introduced FTS, featured transition systems, a formalism designed to describe the combined behaviour of a whole system family. While allowing to model very detailed behavioural variations, FTS leverages on treating features as first-class abstractions and supports separation of concerns. A tool-supported model checking technique allows to verify FTS against temporal properties. Thereby, we can verify all the products of a family at once and pinpoint the products that violate properties. An empirical evaluation showed substantial gains over individual product verification. The source code of the implemented Haskell library is freely available. Integration of our formal machinery with software engineering languages and tools is under way.

Acknowledgements. We thank the anonymous referees for their helpful comments. This work was funded by the FNRS, the Interuniversity Attraction Poles Programme of the Belgian State, Belgian Science Policy (MoVES project) and the Belgian National Bank.

9. REFERENCES

[1] D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel. Learning operational requirements from goal models. In *ICSE 31*, pages 265–275, 2009.

[2] P. Asirelli, M. H. ter Beek, S. Gnesi, and A. Fantechi. Deontic logics for modeling behavioural variability. In *VaMoS'09*, pages 71–76, 2009.

[3] F. Bachmann, M. Goedicke, J. C. S. do Prado Leite, R. L. Nord, K. Pohl, B. Ramesh, and A. Vilbig. A meta-model for representing variability in product family development. In *Int. Workshop on Product Family Engineering (PPE)*, pages 66–80, 2003.

[4] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2007.

[5] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003.

[6] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[7] A. Classen. Modelling with FTS: a collection of illustrative examples. Technical Report P-CS-TR SPLMC-00000001, PReCISE Research Center, University of Namur, Namur, Belgium, 2010. www.info.fundp.ac.be/∼acs/fts.

[8] A. Classen, P. Heymans, and P.-Y. Schobbens. What's in a feature: A requirements engineering perspective. In *FASE'08, Held as Part of ETAPS'08*, volume 4961 of *LNCS*, pages 16–30. Springer, 2008.

[9] A. Classen, P. Heymans, T. T. Tun, and B. Nuseibeh. Towards safer composition. In *ICSE 31, Companion Volume*, pages 227–230. IEEE, 2009.

[10] P. C. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, August 2001.

[11] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Form. Methods Syst. Des.*, 1(2-3):275–288, 1992.

[12] K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *GPCE'05*, pages 422–437, 2005.

[13] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *GPCE '06*, pages 211–220. ACM, 2006.

[14] C. Ebert and C. Jones. Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52, 2009.

[15] E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy (extended abstract). In *FOCS 32*, pages 368–377. IEEE, 1991.

[16] A. Fantechi and S. Gnesi. A behavioural model for product families. In *ESEC-FSE'07, Companion*, pages 521–524. ACM, 2007.

[17] A. Fantechi and S. Gnesi. Formal modeling for product families engineering. In *SPLC 2008*, pages 193–202. IEEE CS, 2008.

[18] D. Fischbein, S. Uchitel, and V. Braberman. A foundation for behavioural conformance in software product line architectures. In *ROSATEA '06, ISSTA 2006 workshop*, pages 39–48. ACM Press, 2006.

[19] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *CAV 2001*, number 2102 in LNCS, pages 53–65, 2001.

[20] A. Gruler, M. Leucker, and K. Scheidemann. Modeling and model checking software product lines. In *IFIP WG 6.1 FMOODS '08*, pages 113–131. Springer, 2008.

[21] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, SEI, CMU, November 1990.

[22] J. Kramer, J. Magee, M. Sloman, and A. Lister. Conic: an integrated approach to distributed computer control systems. *Computers and Digital Techniques, IEE Proceedings E*, 130(1):1–10, 1983.

[23] K. G. Larsen, U. Nyman, and A. Wasowski. Modal i/o automata for interface and product line theories. In *ESOP*, pages 64–79, 2007.

[24] K. Lauenroth, S. Töhning, and K. Pohl. Model checking of domain artifacts in product line engineering. In *IEEE/ACM ASE*, 2009.

[25] H. C. Li, S. Krishnamurthi, and K. Fisler. Verifying cross-cutting features as open systems. In *SIGSOFT FSE*, pages 89–98, 2002.

[26] J. Liu, J. Dehlinger, and R. Lutz. Safety analysis of software product lines using state-based modeling. *J. Syst. Softw.*, 80(11):1879–1892, 2007.

[27] M. Mendonca, A. Wasowski, and K. Czarnecki. SAT-based analysis of feature models is easy. In *SPLC'09*, pages 231–240, 2009.

[28] B. Morin, O. Barais, G. Nain, and J.-M. Jézéquel. Taming dynamically adaptive systems using models and aspects. In *ICSE '09*, pages 122–132. IEEE, 2009.

[29] M. Plath and M. Ryan. Feature integration using a feature construct. *Sci. Comput. Program.*, 41(1):53–84, 2001.

[30] P. Schnoebelen. The complexity of temporal logic model checking. In *Advances in Modal Logic 4*, pages 393–436, 2002.

[31] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Feature Diagrams: A Survey and A Formal Semantics. In *RE'06*, pages 139–148, 2006.

[32] W. M. P. van der Aalst, M. Dumas, F. Gottschalk, A. H. M. ter Hofstede, M. L. Rosa, and J. Mendling. Correctness-preserving configuration of business process models. In *FASE'08, Held as Part of ETAPS'08*, pages 46–61, 2008.

[33] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *LICS'86*, pages 332–344. IEEE CS, 1986.

[34] T. Ziadi, L. Hélouët, and J.-M. Jézéquel. Towards a UML profile for software product lines. In *Int. Workshop on Product Family Engineering (PPE)*, pages 129–139, 2003.