Symbolic Model Checking of Software Product Lines

Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay

Symbolic Model Checking of Software Product Lines Technical Report

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

Axel Legay
IRISA/INRIA Rennes, France and
University of Liège, Belgium
axel.legay@irisa.fr

ABSTRACT

We study the problem of model checking software product line (SPL) behaviours against temporal properties. This is more difficult than for single systems because an SPL with n features yields up to 2^n individual systems to verify. As each individual verification suffers from state explosion, it is crucial to propose efficient formalisms and heuristics.

We recently proposed *featured transition systems* (FTS), a compact representation for SPL behaviour, and defined algorithms for model checking FTS against linear temporal properties. Although they showed to outperform individual system verifications, they still face a state explosion problem as they enumerate and visit system states one by one.

In this paper, we tackle this latter problem by using symbolic representations of the state space. This lead us to consider computation tree logic (CTL) which is supported by the industry-strength symbolic model checker NuSMV. We first lay the foundations for symbolic SPL model checking by defining a feature-oriented version of CTL and its dedicated algorithms. We then describe an implementation that adapts the NuSMV language and tool infrastructure. Finally, we propose theoretical and empirical evaluations of our results. The benchmarks show that for certain properties, our algorithm is over a hundred times faster than model checking each system with the standard algorithm.

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

*FNRS Research Fellow

ICSE '11, May 21–28, 2011, Honolulu, Hawaii, USA
.

1. INTRODUCTION

In *Software Product Line (SPL)* development, systems are designed as families with the goal of making economies of scale through systematic reuse of development artefacts [15]. The different variants of the system (called "products") are identified upfront and a model of their differences and commonalities is created. *Feature Diagrams (FDs)* [23, 35] (see, e.g., Figure 1) are typically used for that, features being atomic units of difference that appear natural to stakeholders and technicians alike [12].

SPLs are commonplace among embedded and critical systems [17]. Formal modelling and verification of SPL behaviour is thus vital for quality assurance. In this paper, we are interested in model checking techniques.

The *model checking problem* consists in deciding whether a system satisfies a given *temporal logic* property. For *Transition Systems (TS)*, a common mathematical representation of system behaviour, there exist various model checking algorithms whose characteristics mainly depend on the logic used to specify the properties. In the literature, the most common logics are *Linear Temporal Logic (LTL)* [33] which allows to reason on single paths, and *Computational Tree Logic (CTL)* [9] which quantifies on sets of paths.

When moving from single systems to SPLs, the model checking problem becomes harder [14]. Scalability issues arise when creating behavioural models of the whole SPL because the number of products is exponential in the number of features. This SPL-specific exponential blowup adds to the more common phenomenon of state space explosion that is incurred when the individual systems are verified. In the SPL context, there are thus two major sources of complexity that make it even more crucial to look for efficient verification formalisms and heuristics.

To address these challenges, we recently proposed *Featured Transition Systems (FTS)* [13], a formalism for specifying behaviour in SPLs. An FTS represents the behaviour of *all* the products of an SPL in a single and compact model. In addition to the introduction of FTS, the major contribution in [13] were efficient algorithms to model check FTS over LTL. Those algorithms exploit the structure of the FTS in order to avoid individual product verification and thereby mitigate one source of state space explosion.

Nevertheless, the algorithms from [13] enumerate and visit system states *one by one*. Although we showed that they perform better than the exhaustive application of standard algorithms to individual products, they still face a state explosion problem. An existing solution to that in single sys-

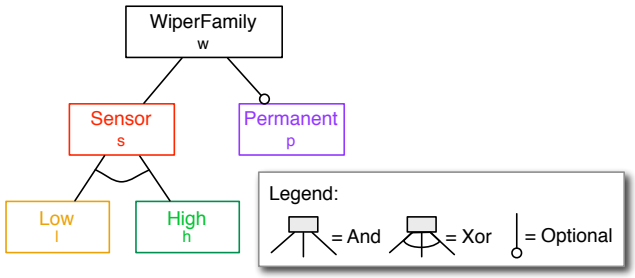


Figure 1: FD of a windscreen wiper controller SPL.

tem model checking is to use symbolic representations [29]. Given a TS, a symbolic representation is a compact data structure for a large set of states and transitions. Symbolic algorithms have been shown to be applicable in many cases where exhaustive techniques do not scale [7].

In this paper, we combine FTS and symbolic model checking to tackle both the aforementioned sources of complexity at once. We propose a symbolic encoding for FTS as well as fully symbolic algorithms. Naturally, this has lead us to consider CTL which is supported by the industry-strength symbolic model checker NuSMV. As a specification language we propose *feature CTL (fCTL)*, an extension of CTL that allows to reason on sets of products. Our algorithm is linear in the size of the state-space of the FTS and clearly outperforms a previously published algorithm [26].

While our algorithmic insights are of interest *per se*, we went further and implemented them as part of NuSMV. As the input language, we use *fSMV*, an existing feature-oriented extension of the NuSMV language [32] which we have shown to be a high-level representation of FTS [10]. The implementation is available at the FTS website [1]. Benchmarks conducted on an elevator system SPL show that our algorithm is up to two orders of magnitude faster than the classical symbolic algorithm executed on each product.

The paper is structured as follows. In Section 2, we recall the basics of TS and FTS. Section 3 introduces the logic fCTL and the corresponding model checking problem. Our symbolic algorithms for model checking fCTL over FTS are presented in Section 4. Section 5 discusses how to implement these algorithms, while Section 6 evaluates them theoretically and empirically. Sections 7 and 8 conclude the paper with an overview of the related work and a conclusion.

2. BACKGROUND

A *Feature Diagram (FD)* is a concise representation for the valid products of an SPL (see, e.g., [23, 35]). Boxes denote features (from a set N) and edges denote decomposition of features. The semantics of an FD d, denoted $[\![d]\!]_{FD}$, is its set of valid *products*, i.e., a set of sets of features: $[\![d]\!]_{FD} \subseteq \mathcal{P}(N)$.

As an example, consider the FD of a windscreen wiping controller SPL (inspired by [22]) presented in Figure 1. The controllers have a rain sensor that is either of high or of low quality, the high quality sensor being able to distinguish between heavy and low rain. There is an optional permanent wiping feature that allows the driver to bypass the sensor. The semantics of this FD would be the following set of four products (using the short feature names):

$$\big\{\{w,s,l\},\{w,s,h\},\{w,s,l,p\},\{w,s,h,p\}\big\}.$$

Note that this is an illustrative example with a few products only. Our industrial partners have FDs with close to 80 features and millions of products [36]. In the literature, SPLs with several thousands of features have been reported [5].

In this paper, behaviours of individual products are represented with *Transition Systems (TS)* [4]. A TS is a directed graph whose transitions are labelled with actions, and whose states are labelled with atomic propositions.

DEFINITION 1 (TRANSITION SYSTEM). *A TS ts is a tuple $ts = (S, Act, trans, I, AP, L)$ 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.*

An *execution* (also called behaviour) of ts is an infinite sequence $\sigma = s_0\alpha_1 s_1\alpha_2 \ldots$ with $s_0 \in I$ such that $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ for all $0 \leq i$. A *path* is an execution from which the information about the transitions has been removed, i.e., the path π for the execution σ is the sequence $s_0 s_1 \ldots$. The ith state in a path π is denoted by π_i, the first state being π_0. The semantics of a TS, written $[\![ts]\!]_{TS}$, is its set of paths.

In [13], we proposed *Featured Transition Systems (FTS)*, a compact model for representing the behaviours of all the products of an SPL. FTSs are TSs in which individual transitions are labeled with features. A transition is part of a product if and only if its feature is part of the product. In this paper, we use a slight generalisation of FTS called FTS$^+$. In FTS$^+$, transitions are labelled with Boolean functions over the features (rather than single features), called *feature expressions*. This allows for greater expressiveness, e.g., the situations in which a transition is present if and only if feature a and not feature b are part of the product can be easily modelled with the feature expression $a \wedge \neg b$. It also has the nice side-effect of simplifying the definitions. Formally:

DEFINITION 2. *An FTS$^+$ is a tuple $fts = (S, Act, trans, I, AP, L, d, \gamma)$, where*

- *$S, Act, trans, I, AP, L$ are defined as in Definition 1,*
- *d is a feature model,*
- *$\gamma : trans \to \big(\{0,1\}^{|N|} \to \{0,1\}\big)$ is a total function, labelling each transition with a* feature expression*, i.e., a Boolean function over the set of features.*

The behaviour of the four products of the wiper SPL can be represented with the FTS$^+$ in Figure 2(a). The feature expression of a transition is shown next to its action label, separated by a slash. In addition, transitions labeled with a single feature are coloured in the same way as the features in Figure 1. The driver controls the wiper with a lever. Without the permanent wiping feature, moving the lever up (①→③) activates the sensor. If the feature is supported, then the first move up (①→②) activates permanent wiping

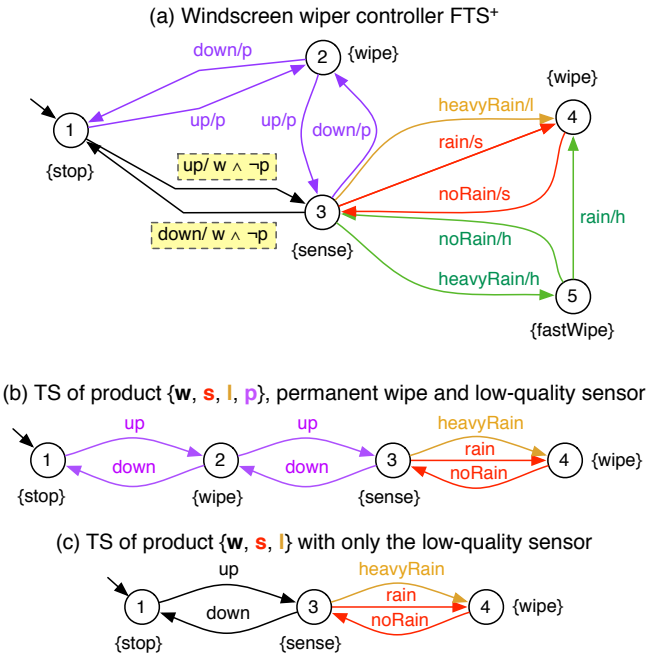


Figure 2: FTS$^+$ of the wiper controller SPL.

and a second move up (②→③) activates the sensor. In the presence of the permanent wiping feature, the first move up leads to state ② rather than ③. This is modelled by the feature expression of transition ①→③, $w \wedge \neg p$, which requires p to be false. Thus, if transition ①→③ is part of the product, transition ①→② cannot be. Moving the lever down deactivates a function in a similar way. In sensing mode, the low quality sensor will react to normal and heavy rain in the same way (moving to state ④). The high quality sensor reacts differently, leading to state ⑤ in heavy rain.

The behaviour of a particular product of the SPL is obtained through *projection*. This operation removes all transitions of the FTS$^+$ whose feature expression does not evaluate to *true* in the product. The result of a projection is a TS. Formally, we have the following definition.

Definition 3 (Projection in FTS$^+$). *The projection of an FTS$^+$ fts to a product $p \in [\![d]\!]$, noted $fts_{|p}$, is the TS $t = (S, Act, trans', I, AP, L)$ where $trans' = \{t \in trans \mid \gamma(t)(f_1 \in p, .., f_n \in p) = 1\}$.*

Figures 2(b) and 2(c) illustrate the projection operation on the FTS$^+$ presented in Figure 2(a). Figure 2(b) is a controller with a low quality rain sensor that supports permanent wiping. It corresponds to the following projection operation: $fts_{|\{w,s,l,p\}}$. Figure 2(c) represents a controller with only the low quality rain sensor, obtained from the FTS$^+$ with the operation: $fts_{|\{w,s,l\}}$.

The FTS$^+$ represents the behaviour of all products of the SPL. Its semantics is thus the union of the standard TS semantics of all possible projections. Formally:

Definition 4 (Semantics of an FTS$^+$).

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

FTS and FTS$^+$ are meant to be the mathematical foundation for SPL behaviour specification. We do not expect engineers to write their specifications in FTS directly. In order to make our techniques more widely applicable, we need to provide high-level languages on top of FTS. This is one of the motivations and contributions of this paper.

3. COMPUTATION TREE LOGIC IN SPLS

In this section, we begin our study of the model checking problem for SPLs. Our objective is to verify *Computation Tree Logic (CTL)* properties for all the products of an SPL. In the case of FTS$^+$ model checking, the CTL formula is evaluated over the computation tree of the FTS$^+$, that is, the computation trees of all products in the SPL. This additional dimension, viz. the products for which a property holds, is missing in the logic. In contrast to our earlier work on LTL [13], we decided to make it explicit and define a version of CTL adapted to SPL verification.

3.1 Encoding Sets of Products

A CTL algorithm computes the set of states that do satisfy the property being checked. When model checking SPLs, however, the notion of satisfaction depends on the products of the SPL. A property is satisfied by a state *for a certain set of products*. Hence, a first challenge is to find an efficient representation for sets of products.

First, observe that representing sets of products explicitly is rather inefficient. In [13], we proposed a symbolic representation which is convenient for LTL model checking of FTS (not FTS$^+$). The idea was to encode a set of products with a set of *required features* rf and a set of *excluded features* ef so that a pair (rf, ef) can be expanded as follows.

Definition 5. *Let N be a set of features. A pair $(rf, ef) \in \mathcal{P}(N) \times \mathcal{P}(N)$ represents a set of products $px \in \mathcal{PP}(N)$ such that $[\![(rf, ef)]\!] \triangleq \{p \in [\![d]\!]_{FD} \mid rf \subseteq p \wedge ef \cap p = \emptyset\}$.*

This representation is less efficient in FTS$^+$ where transitions can be labeled by arbitrary Boolean expressions. Furthermore, as we shall see, CTL algorithms rely on set operations (union, intersection, complementation, etc.) and equivalence checking (as the algorithms generally rely on fixed point calculations). Those operations have to be defined and implemented on the above encoding. A set $\{(\{a, b\}, \{c, d\}) \ldots (\{e, f\}, \{g, h\})\}$ can be viewed as a disjunction of conjunctions of literals (the features), i.e., a formula in disjunctive normal form (DNF): $(a \wedge b \wedge \neg c \wedge \neg d) \vee \ldots \vee (e \wedge f \wedge \neg g \wedge \neg h)$. Unfortunately, intersection, complementation and equivalence checking cannot be implemented efficiently for formulae in DNF. In this paper, we thus propose an alternative approach. The idea is to encode a set of products px by a Boolean function whose literals are defined in terms of features. Formally, we have the following.

Definition 6. *A Boolean function $\chi_{px}(x_1, .., x_n)$ over the set of features N, $\chi_{px} : \{0,1\}^{|N|} \rightarrow \{0,1\}$, is a symbolic encoding of a set of products $px \in \mathcal{PP}(N)$ such that*

$$[\![\chi_{px}]\!] \triangleq \{p \in [\![d]\!]_{FD} | \chi_{px}(x_1, .., x_n) = 1 \text{ with } x_i = 1 \iff f_i \in p \text{ for } i \in [1, n]\}$$

Intuitively, the Boolean function is the characteristic function of the set, returning *true* for elements in the set, and *false* otherwise. Here and in the remainder of the paper, 1 and 0 denote *true* and *false*, respectively. Boolean functions

can be represented symbolically with *Boolean Decision Diagrams* (BDDs) which are often compact and on which the aforementioned operations can be computed efficiently [6]. The semantics of an FD being the set of valid products, it can also be encoded in this way.

3.2 Feature Computation Tree Logic (fCTL)

We propose *feature Computation Tree Logic (fCTL)*, an extension of CTL that makes the product dimension apparent. fCTL is obtained by augmenting CTL formulae with *product quantifiers*, also called *guards*. Those guards are Boolean functions (as in Definition 6) that specify for which products a CTL formula should hold. The formula is trivially satisfied for all other products. Formulae without a product quantifier (and thus pure CTL) implicitly range over all products. The *syntax* of fCTL is given as follows:

DEFINITION 7. *An fCTL formula ϕ is an expression*

$$\begin{aligned}\phi ::= \quad & 1 \mid a \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \\ & [\chi_{px}]EX\phi \mid [\chi_{px}]AX\phi \\ & [\chi_{px}]E(\phi_1 U \phi_2) \mid [\chi_{px}]A(\phi_1 U \phi_2) \\ & [\chi_{px}]EG\phi \mid [\chi_{px}]AG\phi \\ & [\chi_{px}]AF\phi \mid [\chi_{px}]AF\phi\end{aligned}$$

with $a \in AP$ and $px \subseteq \mathcal{P}(N)$.

As in CTL, E (resp. A) means at least one path satisfies (resp. all paths satisfy) the following path operator. The path operators are: *next*, $X\phi$ requires the next state to satisfy ϕ; *until*, $\phi_1 U \phi_2$ requires ϕ_1 to hold until ϕ_2 is satisfied (and ϕ_2 has to be satisfied at some point); *finally*, $F\phi$ requires ϕ to hold in some future state; and *generally*, $G\phi$ requires ϕ to hold for all future states. It is well-known that any CTL formula can be expressed in existential normal form (ENF) [4], that is, only with path operators EX, EU and EG. This observation naturally extends to fCTL.

An fCTL formula is evaluated over the executions of an FTS (or an FTS$^+$). Its *semantics* is defined as follows.

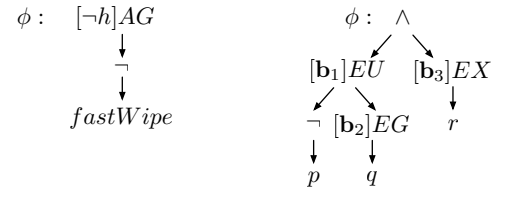
DEFINITION 8. *Semantics of an fCTL formula wrt. an FTS$^+$ fts, a state $s \in S$, and a set of products $px \subseteq \mathcal{P}(N)$.*

$$s, px \models \phi \iff \forall p \in px \bullet s, p \models \phi$$

Where N is the set of features, $p \subseteq N$ is a product and satisfaction of a formula ϕ in a state s and in a product p.

$$\begin{aligned}
&s,p \models a && \text{iff } a \in L(s)\\
&s,p \models \neg\phi && \text{iff } \neg(s,p \models \phi)\\
&s,p \models \phi_1 \wedge \phi_2 && \text{iff } (s,p \models \phi_1) \wedge (s,p \models \phi_2)\\
&s,p \models [\chi_{px}]EX\phi && \text{iff } p \in [\![\chi_{px}]\!] \Rightarrow \exists \pi \in [\![fts_{\,|p}]\!]_{FTS}\\
&&& \quad \bullet\ \pi_0 = s \wedge (\pi_1, p \models \phi)\\
&s,p \models [\chi_{px}]E(\phi_1 U \phi_2) && \text{iff } p \in [\![\chi_{px}]\!] \Rightarrow \exists \pi \in [\![fts_{\,|p}]\!]_{FTS}\\
&&& \quad \bullet\ \pi_0 = s \wedge (\exists j \geq 0 \bullet (\pi_j, p \models \phi_2)\\
&&& \quad\quad \wedge \forall 0 \geq i < j \bullet (\pi_i, p \models \phi_1))\\
&s,p \models [\chi_{px}]EG\phi && \text{iff } p \in [\![\chi_{px}]\!] \Rightarrow \exists \pi \in [\![fts_{\,|p}]\!]_{FTS}\\
&&& \quad \bullet\ \pi_0 = s \wedge \forall i \geq 0 \bullet (\pi_i, p \models \phi))
\end{aligned}$$

Note that existential quantification in fCTL requires that at least one path satisfies the formula in *all* products (not in just one product). If all the guards of an fCTL formula are *true*, then the semantics of the formula is equivalent to the standard CTL semantics. Observe also that a formula can be trivially satisfied: if a guard is unsatisfiable, the quantification ranges over an empty set of products making it trivially satisfied.



(a) $[\neg h]AG\ \neg fastWipe$ (b) $[\mathbf{b}_1]E\big(\neg p\, U([\mathbf{b}_2]EGq)\big) \wedge [\mathbf{b}_3]EXr$

Figure 3: Examples of parse trees.

An example property for the wiper FTS$^+$ in Figure 2 is

$$[\neg h]AG\ \neg fastWipe$$

which expresses that *"in absence of the high quality sensor feature (h), it is impossible to activate fast wipe"*. This property is satisfied by the model, since the transition leading to the fast wipe state is not present if the high quality sensor is not included.

In general, an FTS$^+$ satisfies an fCTL formula if all its initial states satisfy the formula for all products. This leads to the following *satisfaction relation* and associated model checking problem for fCTL over FTS$^+$.

DEFINITION 9. $fts \models \phi \iff \forall i \in I \bullet i, [\![d]\!]_{FD} \models \phi$

DEFINITION 10 ($\mathrm{MC}_{\mathrm{CTL}}(\mathrm{FTS}, \phi)$). *Given a property ϕ and an FTS$^+$ fts, $MC_{CTL}(fts, \phi)$ returns* true *iff $fts \models \phi$. If $fts \not\models \phi$, it returns* false *and a Boolean function χ_{px} which identifies the products that violate the property, i.e., such that $\exists i \in I \bullet i, [\![\chi_{px}]\!] \not\models \phi$ and $\forall i \in I \bullet i, [\![\neg\chi_{px}]\!] \models \phi$.*

In the following section, we propose efficient algorithms for solving this problem.

4. MODEL CHECKING FCTL ON FTS$^+$

Like any CTL algorithm, our fCTL algorithm uses the parse tree of a formula to decompose it into subproblems that can be handled independently.

DEFINITION 11. *Given an fCTL formula ϕ, its* parse tree *is a graph, with ϕ as the root, where leaf nodes correspond to terminal formulae (i.e., 1 or a) and where each intermediate node corresponds to a production of Definition 7 with its children being the subformulae of this production.*

As an example, the parse tree of the aforementioned property is given in Figure 3(a). A more complicated example is shown in Figure 3(b). The model checking algorithm traverses the parse tree of the formula bottom-up. First, the states satisfying the formulae of the leaves are computed,[1] then the information is used to compute the states satisfying the formulae of their parents and so on. The last step is to compute the states satisfying the whole formula ϕ.

In Section 4.1, we propose a symbolic encoding of FTS$^+$. Based on this encoding, we discuss the fundamental operations and concepts in Section 4.2 and define algorithms to model check basic fCTL formulae in Section 4.3.

[1]In what follows we shall see that our algorithms also compute the set of products for which the CTL formula obtained by removing the guard of the fCTL formula is satisfied.

4.1 Symbolic Encoding of FTS$^+$

The main motivation for this work is that symbolic algorithms can, to some extent, address the state explosion problem and allow to verify large state spaces [29]. In the symbolic setting, sets of states and the transition relation are encoded directly with their characteristic functions.

Let us first fix some basic notation. As a starting point we assume the existence of a binary encoding of states, that is, a function $enc : S \to \{0,1\}^k$, where k is chosen large enough to encode all states. Products can be encoded as bit vectors of size n, n being the number of features. In the rest of the paper, we also use enc to note encoded products. Note that with this encoding, $\{0,1\}^k$ implicitly denotes the sets of all (encoded) states and $\{0,1\}^n$ the set of all (encoded) products. Any subset of states $T \subseteq S$ can be represented by its characteristic function, χ_T, that is

$$\chi_T(\overline{s}) : \{0,1\}^k \to \{0,1\} \bullet \chi_T(enc(s)) = 1 \iff s \in T$$

χ stands for *characteristic function* [4]. The subscript of χ, e.g., '$X \cup Y$' in $\chi_{X\cup Y}$, denotes the set for which this is the characteristic function. In parentheses follow the variables on which the function is defined. By convention, $\overline{s}$ (resp. $\overline{p}$) denotes a vector of variables encoding a state (resp. product). The *cofactor* $\chi_{T[\overline{s}\leftarrow enc(x)]}$ of a Boolean function $\chi_T(\overline{s},\overline{p},\ldots)$ is the Boolean function over the variables $\overline{p},\ldots$ obtained by replacing the variables $\overline{s}$ by the value they take in $enc(x)$. In the implementation, each $\chi(x_1,..,x_k)$ becomes a BDD over variables $x_1,..,x_k$.

With the notation in place, we now show how an FTS$^+$ can be encoded symbolically. The set of states is represented by χ_S and the set of initial states by χ_I. As usual, the labelling of states with atomic propositions L is represented by recording for each atomic proposition $a \in AP$ the set of states χ_a that are labeled by the proposition:

$$\chi_a(\overline{s}) : \{0,1\}^k \to \{0,1\} \bullet \chi_a(enc(s)) = 1 \iff a \in L(s).$$

The transition relation is represented by a function that takes two encoded states (start and end) and an encoded product, and returns 1 iff some transition $s \xrightarrow{\alpha} s'$ exists in the product. The feature expression of a transition is implicitly embedded in the encoding.[2] Formally,

$$\chi_{trans}(\overline{s},\overline{s}',\overline{p}) : \{0,1\}^k \times \{0,1\}^k \times \{0,1\}^n \to \{0,1\},$$

such that $\chi_{trans}(enc(s),enc(s'),enc(p)) = 1$ iff some $s \xrightarrow{\alpha} s'$ in $fts_{|p}$. The feature expression on the transition is the cofactor for the encoding of both states:

$$\chi_{\bigvee_\alpha \gamma(s\xrightarrow{\alpha}s')}(\overline{p}) : \{0,1\}^n \to \{0,1\} \triangleq \chi_{trans[\overline{s}\leftarrow enc(s),\overline{s}'\leftarrow enc(s')]}(\overline{p}).$$

Transitions with the same start and end states are implicitly merged (with a disjunction of their Boolean function labels).

This yields a symbolic encoding for FTS$^+$ covering all of Definition 2.

4.2 Fundamentals of fCTL Model Checking

The fCTL algorithm does a bottom-up traversal of the parse tree. For each subformula ϕ in a given node, it records the set of states and the products, $Sat(\phi)$, that satisfy the subformula. These sets are referred to as *satisfaction sets*. To complete the model checking algorithm for fCTL, it is thus sufficient to give a recursive definition of $Sat(\phi)$. Formally, a satisfaction set $Sat(\phi)$ is a set of couples:

$$Sat(\phi) \subseteq S \times \mathcal{P}(N)$$

where N is the set of features, so that $(s,p) \in Sat(\phi)$ means that $s,p \models \phi$ according to Definition 8. Such a satisfaction set is also encoded by its characteristic function,

$$\chi_{Sat(\phi)}(\overline{s},\overline{p}) : \{0,1\}^k \times \{0,1\}^n \to \{0,1\},$$

so that $\chi_{Sat(\phi)}(enc(s),enc(p)) = 1$ iff $s,p \models \phi$.

Most CTL model checking algorithms rely on fixed point computations. As we shall see in the next subsection, this remains the same for fCTL algorithms. The basis of our fixed point algorithms is an operator that, given a state s and a product p, returns the predecessors of s in p. Intuitively, the predecessors of a state s in a product p are the states s' that can reach s via a transition $t = s' \xrightarrow{\alpha} s$ so that p satisfies the feature expression of the transition $p \in [\![\gamma(t)]\!]$.

For instance, consider state ③ in Figure 2(a). Its predecessors in the product $\{w,s,l,p\}$, i.e., $Pre(③,\{w,\ s,\ l,\ p\})$, are states ② and ④. This is easy to see in Figure 2(b). Its predecessors in the product $\{w,s,l\}$ are states ① and ④. This information is contained literally in the transition relation:

$$\chi_{Pre(s,p)}(\overline{x}) : \{0,1\}^k \to \{0,1\} \triangleq \chi_{trans[\overline{s}'\leftarrow enc(s),\overline{p}\leftarrow enc(p)]}(\overline{x}),$$

that is, the cofactor of the transition relation χ_{trans} for p and s as target state.

Of course, this calculation has to be very efficient since it is executed at each step of the algorithm. Therefore, the computation cannot rely on single state/product predecessor computations to accomplish this. We rather need to compute it on *a set* of such couples, generally a satisfaction set of some property ϕ. This leads us to define the operator $SetPre$ as follows.

Definition 12.

$$\chi_{SetPre(Sat(\phi))}(\overline{s},\overline{p}) : \{0,1\}^k \times \{0,1\}^n \to \{0,1\} \triangleq \exists \overline{s}' \bullet \chi_{Sat(\phi)}(\overline{s}',\overline{p}) \wedge \chi_{trans}(\overline{s},\overline{s}',\overline{p}).$$

Intuitively, $SetPre(Sat(\phi))$ is the set of couples (s,p) such that there exists a state s' that satisfies ϕ in product p and to which s has a transition in product p. Since the operation is computed on the symbolic encoding of the sets, it does not consider states or products individually.

4.3 Algorithms

Having the fundamentals covered, we can proceed to the model checking algorithms. The satisfaction sets for fCTL state formulae are recursively defined as follows.

Definition 13. *fCTL state formulae satisfaction sets:*

$$\begin{aligned}
\chi_{Sat(true)}(\overline{s},\overline{p}) &= 1\\
\chi_{Sat(a)}(\overline{s},\overline{p}) &= \chi_a(\overline{s})\\
\chi_{Sat(\phi_1\wedge\phi_2)}(\overline{s},\overline{p}) &= \chi_{Sat(\phi_1)}(\overline{s},\overline{p}) \wedge \chi_{Sat(\phi_2)}(\overline{s},\overline{p})\\
\chi_{Sat(\neg\phi)}(\overline{s},\overline{p}) &= \neg\chi_{Sat(\phi_1)}(\overline{s},\overline{p})
\end{aligned}$$

These rules are straightforward and should be intuitive. Let us now study satisfaction sets for fCTL path formulae.

EX.

This definition being recursive, $Sat(\phi)$ is supposed to be known. To obtain $Sat(EX\phi)$ it is sufficient to calculate the predecessors of $Sat(\phi)$, that is, to apply the $SetPre$ operator from Definition 20 to $Sat(\phi)$.

[2] Which is natural as both the transitions and the feature expression are Boolean functions.

DEFINITION 14. $\chi_{Sat(EX\phi)}(\overline{s},\overline{p}) \triangleq SetPre(Sat(\phi))(\overline{s},\overline{p})$

EU.

In standard CTL model checking, $E(\phi_1 U \phi_2)$ can be computed with the smallest fixed point: $\mu T \bullet \phi_2 \vee (\phi_1 \wedge EX\ T)$. The corresponding algorithm starts with the states satisfying ϕ_2 and then searches backwards for all predecessors satisfying ϕ_1. The procedure for fCTL model checking behaves in a similar way.

According to Tarski's fixed point theorem, the above fixed point can be computed as follows:

DEFINITION 15. $\chi_{Sat(E(\phi_1 U \phi_2)} = \chi_{T_i} \bullet \forall j > i \bullet \chi_{T_j} = \chi_{T_i}$
where $\chi_{T_0}(\overline{s},\overline{p}) = \chi_{Sat(\phi_2)}(\overline{s},\overline{p})$
$\chi_{T_{i+1}}(\overline{s},\overline{p}) = \chi_{T_i}(\overline{s},\overline{p}) \vee \big(\chi_{Sat(\phi_1)}(\overline{s},\overline{p}) \wedge \chi_{SetPre(T_i)}(\overline{s},\overline{p}) \wedge \neg\chi_{T_i}(\overline{s},\overline{p})\big)$

In each iteration, we add the states $(\overline{s},\overline{p})$ that satisfy ϕ_1, i.e. $\chi_{Sat(\phi_1)}(\overline{s},\overline{p})$, and are predecessors of a state in T_i, i.e. $\chi_{SetPre(T_i)}(\overline{s},\overline{p})$. An optimisation known in current CTL algorithms, and crucial here, is to only add states that were not already in T_i, i.e. $\neg\chi_{T_i}(\overline{s},\overline{p})$. Otherwise, previously visited states would be re-visited, which would be inefficient especially due to the added feature variables.

EG.

The algorithm for EG is similar to the previous case, except that the greatest fixed point, $\nu T \bullet \phi \wedge EX\ T$, has to be computed. This fixed point can be calculated as follows:

DEFINITION 16. $\chi_{Sat(EG\phi)} = \chi_{T_i} \bullet \forall j > i \bullet \chi_{T_j} = \chi_{T_i}$
where $\chi_{T_0}(\overline{s},\overline{p}) = \chi_{Sat(\phi)}(\overline{s},\overline{p})$
$\chi_{T_{i+1}}(\overline{s},\overline{p}) = \chi_{T_i}(\overline{s},\overline{p}) \wedge \chi_{SetPre(T_i)}(\overline{s},\overline{p})$

The procedure starts with a large set, viz. all states satisfying ϕ, and progressively shrinks it. At each iteration, only states that are the predecessor of some state in T_i are kept.

Product quantification.

The quantified formula $[\chi_{px}]Sat(\phi)$ is trivially satisfied by all states for the products not in px. For the other products, it is only satisfied if $Sat(\phi)$ is satisfied. The quantified formula can thus be obtained from $Sat(\phi)$ by the following operation.

DEFINITION 17. *Quantifying a satisfaction set over* χ_{px}*:*

$$\chi_{[\chi_{px}]Sat(\phi)}(\overline{s},\overline{p}) = \neg\chi_{px}(\overline{p}) \vee \chi_{Sat(\phi)}(\overline{s},\overline{p})$$

Note that this operation can be implemented as defined here, that is, independently from the calculation of $Sat(\phi)$. This has the advantage that the satisfaction sets for formulae, that are identical except for their guards, will be computed only once. Alternatively, the guard could be used to reduce the state space before calculating $Sat(\phi)$.

Checking the validity of a formula.

The final step of the fCTL model checking algorithm is to check whether all initial states satisfy ϕ, and for which products they do. Given $\chi_{Sat(\phi)}(\overline{s},\overline{p})$, the set of products that violate ϕ is obtained by intersecting the complement of $Sat(\phi)$ with the set of initial states, and then projecting on the state variables. This leaves a Boolean function over the feature variables characterising the set of violating products:

DEFINITION 18. *The set of products* $\chi_{px_{bad}}$ *violating an fCTL property* ϕ *is* $\chi_{px_{bad}}(\overline{p}) = \exists\overline{s} \bullet \chi_I(\overline{s}) \wedge \neg\chi_{Sat(\phi)}(\overline{s},\overline{p})$.

If $\chi_{px_{bad}} = 0$, the property is satisfied by all products. More generally, checking whether a specific set of products $\chi_{px'}$, e.g. the set of valid products in the FD, satisfies ϕ, amounts to testing whether $\chi_{px'}$ implies $\chi_{px_{bad}}$ (meaning that the former is a subset of the latter).

The algorithms for calculating satisfaction sets and quantification, combined with the parse tree computation lead to a complete model checking algorithm for fCTL over FTS$^+$.

THEOREM 19. *Our algorithms compute the satisfaction relation specified in Definition 9. Furthermore the extended validity check provides the information on violating products required in Definition 10.*

5. FROM THEORY TO PRACTICE

The previous section provides a mathematical foundation for symbolic FTS$^+$ model checking. To make our approach available to engineers, we decided to customise the state-of-the-art symbolic model checker NuSMV.[3] Thereby, we can take advantage of its existing infrastructure. We proceed in two steps. The first (Section 5.1) is to define a high-level language that maps to FTS$^+$ and that is close to NuSMV's original input language. The second (Section 5.2) is to implement our algorithms in NuSMV. In Section 5.3, we briefly describe the resulting toolset and its usage.

5.1 A High-Level Language

To specify SPL behaviour, we propose to reuse a language by Plath and Ryan [32]. The language, which we call *fSMV*, is a feature-oriented extension of the input language of NuSMV. A NuSMV model consists of a set of variable declarations which define the state space. For each variable, the value in the next state is defined as a function of the value in the present state through assignments. From this it is straightforward to derive the BDD of the transition relation. The basic wiper example from Section 2 could be expressed in NuSMV as follows:

```
MODULE main
VAR    rain:   boolean;
       wiper:  {on, off};
ASSIGN init(wiper) := off;
       next(wiper) := case rain = 1: on;
                           1:        off;
                      esac;
```

The rain is controlled by the environment and thus modelled as a non-deterministic Boolean variable. The wiper is initially off. As expected, when it rains the wiper is switched on. Otherwise (`1:` can be read as `else:`) it is switched off.

The fSMV language is based on *superimposition* [21]. A feature basically describes the changes it makes to a NuSMV model. It can introduce new variables into the system, override the definition of existing variables and change the values of variables the moment they are read. For example, the permanent wipe feature adds a switch that lets the driver switch on the wiper manually. The switch is modelled with a new non deterministic variable `forced`. The system is changed in such a way that permanent wiping overrides the rain sensor. In fSMV, this yields:

[3] `http://nusmv.irst.itc.it`

```
FEATURE permanent
INTRODUCE
   VAR forced: boolean;
CHANGE
   IF forced = 1 THEN IMPOSE next(wiper) := on;
```

In a companion technical report [10], we have formally established that fSMV and FTS$^+$ are expressively equivalent. This result provides algorithms for translating fSMV into explicit-state FTS$^+$. In order to implement the algorithms of Section 4, we need a translation from fSMV to the encoding defined in Section 4.1.

5.2 Implementing our Algorithms

Plath and Ryan [32] specify how a list of fSMV features is composed with a NuSMV model, the base system, to produce the NuSMV model of a product. This results in a model that only describes the product, and contains no more information on the features it possesses. In particular, it does not match the symbolic encoding of Section 4.1 which requires a single model for all products, and where transitions hold information about the variability of products.

We thus propose a different way of composing fSMV features which creates a single model for all products, and where information about the features is recorded in the states. Although this does not exactly correspond to the encoding of Section 4.1, where this information is kept in the transitions, it has the advantage of being implementable in NuSMV without drastic changes to the model checker or its input language. To begin with, our composition adds a Boolean state variable for each feature. Each transition thus has two copies of the feature variables, instead of one. Formally

$$\chi_{trans}(\overline{s},\overline{p},\overline{s}',\overline{p}') : \{0,1\}^k \times \{0,1\}^n \times \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\},$$

such that $\chi_{trans}(enc(s), enc(p), enc(s'), enc(p)) = 1$ iff some $s \xrightarrow{\alpha} s'$ in $fts_{|p}$. The feature variables are thus left unchanged by the transitions. The initial states of the model are modified to include all possible feature combinations.

With this, the predecessor calculation becomes

Definition 20.

$$\chi_{SetPre(Sat(\phi))}(\overline{s},\overline{p}) : \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\} \\ \triangleq \exists \overline{s}' \bullet \chi_{Sat(\phi)}(\overline{s}',\overline{p}) \wedge \chi_{trans}(\overline{s},\overline{p},\overline{s}',\overline{p}).$$

Observe that this definition coincides with the definition of the predecessors in standard CTL model checking algorithms, under the condition that the feature variables in a transition do not change. Since this is guaranteed by our composition mechanism, we can reuse the standard predecessor calculation implemented in NuSMV. Observe further that the algorithms for calculating satisfaction sets (Section 4.3) all treat state and feature variables indifferently. These observations combined mean that the small change in the encoding allows us to reuse the satisfaction set calculation algorithms of NuSMV.

The final step of the algorithm, i.e., checking whether all initial states satisfy the property, *does* treat feature and state variables differently. It thus needs to be adapted as described in Definition 18, by quantifying away the state variables in order to obtain the set of violating products. If this is not done, the algorithm will just yield $false$ if there are violating products, without indicating which products are to blame.

The modified encoding has another advantage: it allows to express fCTL properties in CTL. This is due to the fact that feature variables are state variables that can be referenced in a property. A set of products χ_{px} can thus be expressed in the specification language, which means that the fCTL formula $[\chi_{px}]\phi$ can be translated to the CTL formula $(\chi_{px}) \implies \phi$.

Our composition mechanism and a small change in the symbolic encoding of FTS$^+$ thus allow us to reduce most of the fCTL model checking algorithms over FTS$^+$ to CTL model checking in NuSMV. The composition mechanism uses a technique similar to [34]; details are provided in [10]. In essence, composition is performed as prescribed by [32], with the addition that all changes made by features are *guarded* by the newly introduced feature variables.

5.3 Toolset

Our toolset is based on the NuSMV model checker. Essentially, it takes as input a list of features specified in fSMV, a NuSMV model which serves as the base system, and one or more fCTL properties. For each property, it determines the products for which it is satisfied and violated.

A first tool is used to compose a feature and a NuSMV model to create a new NuSMV model, which can either be composed with another feature or be verified in NuSMV. Several calls can be chained with pipes. For benchmarking purposes, this tool implements both the composition mechanism from [32] and our modified version thereof [10].

The result of this composition is then passed on to the modified version of NuSMV. It has a new command line switch `-fbdd` which activates the changes discussed in the previous section, meaning that NuSMV will print for each violated property a Boolean expression characterising the products that violate the property. The modifications made to NuSMV are available as a patch for NuSMV 2.5.0.

Two additional tools provide (i) a filter that analyses the NuSMV output and produces an overview of the results, (ii) the ability to add preprocessed quantifiers to NuSMV files, making it possible to parameterise models (e.g. the number of floors in an elevator SPL model).

All of this is available (open source) at the FTS website [1].

6. EVALUATION

6.1 Theoretical Evaluation

Our algorithm for model checking of fCTL over FTS$^+$ is $O(|S|.|\phi|.2^n)$ where S is the set of states and n the number of features. Basically, a satisfaction set is calculated for each node in the formula giving the factor $|\phi|$. This calculation is linear in the size of the state space for $Sat(1)$, $Sat(a)$, $Sat(\neg\phi)$, $Sat(\phi_1 \wedge \phi_2)$ and $Sat(EX\phi)$. The fixed points of $Sat(E(\phi_1 U \phi_2))$ and $Sat(EG\phi)$ both take $O(|S|.2^n)$ since, in the worst case, they proceed monotonically through 2^n products for each state.

For reference, single-system CTL model checking of TSs has a computational complexity of $O(|S|.|\phi|)$. The additional exponential factor of our algorithm cannot be avoided unless model checking is restricted to models less powerful than FTS$^+$, as done in [27]. Also note that our algorithm remains linear in the size of the state space and is more efficient than the one presented in [26]. More precisely, the latter is $O(|\phi|.|S|!) = O(|\phi|.|S|^{|S|})$. The algorithm of [26] is thus in EXPTIME whereas ours is in E, i.e. DTIME($2^{O(x)}$), a class that *"captures a more benign aspect of exponential time"* [31]. Furthermore, it is important to note that in

Table 1: Benchmark results for the elevator system with six floors (times are in seconds).

Property	Value	Enum.	Single	Speedup
01	*false*	44.00	1.05	41.90
01'	*true*	34.02	0.13	261.69
04	*false*	67.76	18.44	3.67
02	*false*	52.36	1.87	28.00
03a	*false*	76.67	22.42	3.42
03b	*false*	77.98	27.21	2.87
05a	*false*	105.07	322.53	0.33
05b	*true*	30.67	0.04	766.75
05-part	*true*	54.63	0.32	170.72
05c	*false*	88.63	78.36	1.13
05d	*true*	30.93	0.05	618.60
05e	*false*	67.45	18.39	3.67
05'	*false*	131.78	63.61	2.07
06	*true*	68.36	20.42	3.35
07	*true*	73.06	36.89	1.98

practice, the size of the state space is much larger than the number of features.

6.2 Empirical Evaluation

In [32], the authors propose the fSMV language along with a verification technique for CTL that is based on an exhaustive enumeration of the set of products (although they limit themselves to couples of features). Such a product-enumerative approach is exactly what our algorithms intend to avoid. While both approaches produce equivalent results, we argue that model checking a single model with variability (i.e., the model of the whole SPL) is in general more efficient. Experiments with our earlier LTL algorithms also suggest that [13]. Here, we test this hypothesis in the symbolic context through benchmarks that compare the runtime of product-enumerative vs. single-model approaches.

To this end, we used the elevator system by Plath and Ryan [32]. We extended the fSMV models provided with the original paper in two ways. First, we made the number of floors (initially fixed at five) variable via preprocessed quantifiers. Secondly, we added four more features to the system, resulting in a total of nine features. All features are independently optional, which means that there are 2^9 products. Benchmarks were run for the fifteen properties of the base system (mostly combined safety and liveness properties) with the number of floors ranging from four to eight. Table 1 lists runtimes (in seconds) and speedups for the case of six floors. The benchmarks compare (for each property)

- the total runtime of 2^9 model checks that enumerate all products explicitly (column 'Enum.' in Table 1);
- the runtime of a single NuSMV model check following our method (column 'Single' in Table 1).

Each property was benchmarked individually. The property number reported in the Table 1 refers to the number given to the property in the NuSMV code. The benchmarks were run on an Ubuntu machine with an Intel Core2 Duo at 2.80 GHz with 4 Gb of RAM.

The size of the NuSMV model of the product with all features ranges from 2^{17} states for four floors, to 2^{27} states for eight floors. These are the upper bounds for the size

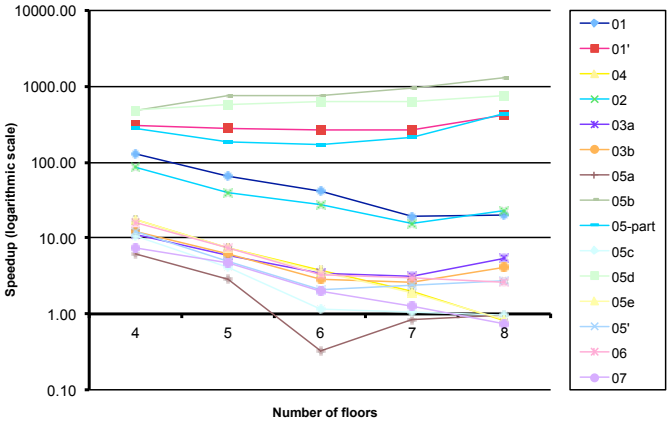


Figure 4: Evolution of speedup with the number of floors (logarithmic scale).

of the models analysed in the *enumerative* benchmarks. As explained earlier, our algorithm only needs one check, but requires an additional variable for each feature. Its models are thus much larger, from 2^{26} states to 2^{36}. The models are distributed with the toolset and available online [1].

An important factor in BDD based model checking is the variable ordering. In order to avoid computing static variable orderings and still be efficient, NuSMV has the parameter `-dynamic`, which causes the BDD package to reorder the variables during verification in case the BDD size grows beyond a certain threshold. While this method works well on small to medium models (up to six floors), its limitations become more and more apparent as the size of the models grows. For eight floors, NuSMV would spend more time reordering variables than actually verifying the property.

In consequence, we computed variable orderings for each number of floors, and used these in all subsequent benchmarks. The model checks of the *single* approach were run with parameters `-df -i orderfile`. Those of the *enumerative* approach were run with `-df -dynamic`. It is important to note that the variable orderings computed for the *single* approach cannot be reused for the *enumerative* case. This is due to the fact that the *enumerative* approach produces 2^9 models with different sets of variables, which would require 2^9 variable orderings. However, due to the absence of the nine feature variables, the individual models of the *enumerative* cases are much smaller than the single model in the *single* case. Therefore, the dynamic variable ordering, while being the only option, should still be rather efficient.

The results show that our approach achieves order-of-magnitude speedups over the enumerative approach. These observations are reported for each property in Figure 4, where we show how speedup evolves when the number of floors grows. Three clusters appear: four high outliers, with speedups greater than 250 and up to 1000; five low outliers with speedups below two or three and sometimes negative; and six stable properties with speedups around ten. A trend that we observed is that with an increasing number of floors, the outliers tend to become more extreme (the high speedups grow, the low speedups descend). This is most likely due to the importance of the static variable ordering for larger models, although we cannot exclude other factors.

In order to limit bias, we went to great lengths to ensure that the *enumerative* benchmarks were as efficient as possi-

ble. For instance, the computation of the 2^9 feature compositions (to create the files that were model checked) for each property was not included in the runtime. Furthermore, the large volume of log files from these runs was cleaned after each run since it would slow down model checking after several runs (because of huge inode lists in the parent folder).

7. RELATED WORK

Let us first compare this paper with our previous work. In [13, 11] we materialised a plan sketched in [14] and proposed FTS as the foundation for behavioural specification and verification of SPLs. The focus of the papers were the formalism and an explicit-state algorithm for omega-regular (e.g., LTL) properties. The present paper extends these results in several ways. We study symbolic, rather than explicit, algorithms. This results in new algorithmic insights and an adapted temporal logic, fCTL. We will not enter into the LTL vs. CTL debate here, but just mention that it extends to SPLs and that one could also add product quantifiers to LTL. Regarding complexity, the LTL and CTL algorithms are rather similar: the complexity of the single-system algorithm multiplied by an exponential factor. However, the CTL algorithm is fully symbolic, it can thus handle much larger state spaces. We further propose a front-end to our theoretical results that is built around the industry-strength NuSMV model checker.

Verification of SPLs. We are not aware of any symbolic model checking approach adapted specifically to SPLs, except for the straightforward product-enumerative approach, which the benchmarks of Section 6 have shown to be largely inefficient compared to our algorithms.

Lauenroth *et al.* [26] propose an explicit-state CTL model checking algorithm for automata labeled with features. There are a number of differences between their approach and ours. A first difference is that their algorithms are not symbolic. Also, while similar to FTS, their modelling language does not allow to label transitions with arbitrary Boolean expressions, and uses a non-standard definition of the parallel composition (which adds transitions that were not in the original automata). Furthermore, the algorithms they propose do not attempt to explore the state space in an efficient manner. Instead, they explore every possible path of the state space which results in an algorithmic complexity that is exponential in the size of the (already huge) state space (see Section 6.1 for details). Since the translation of an automaton accepted by their method into an FTS can be easily achieved in linear time, our algorithms can treat their problems more efficiently.

Not specifically aimed at SPLs, but still comparable, is the compositional approach for CTL model checking of features proposed by Li *et al.* [27]. 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.

In the domain of process calculi, Gruler *et al.* [22] propose PL-CCS, a variant of CCS extended with a product line variant operator that allows to model an alternative choice between two processes. Their model checking procedure is, however, only sketched and, as far as we know, no implementation is available. Moreover, as they introduce a new operator that changes the semantics of the calculus rather drastically, they cannot easily adapt existing tools for their approach (as we do in Section 5).

Behavioural modelling of SPLs. There are approaches for behavioural modelling of SPLs, but that do not provide mechanisms for the verification of temporal properties.

In [25], Larsen *et al.* proposed modal I/O automata to model SPLs whose products are open systems. Their main contribution is to study refinement between two SPLs as well as a method to decompose an SPL. Hence, contrary to us, they do not study procedures for model checking against arbitrary temporal properties.

In [20], Fischbein *et al.* suggest to use modal transition systems (MTS) [24] to model SPLs. A modal TS is a TS equipped with *must* and *may* transitions. *May* transitions are used to introduce variability in the model. The authors are only interested in the notion of behavioural conformance, i.e., deciding whether a behaviour is part of an SPL. They do not consider the model checking problem.

Fantechi and Gnesi [19, 18] introduce variability operators into MTS that allow to specify cases in which $i..j$ outgoing transitions may be taken. This does not overcome the inherent MTS limitation that all may transitions are independent.

Asirelli *et al.* propose MHML, a logic that can express behavioural properties and constraints over features [3, 2]. As an MTS does not explicitly refer to a feature model, the presented algorithm for model checking MHML over MTS cannot be used to find the products that violate a property.

High-level languages for SPLs. Existing high-level languages for modelling SPL behaviour, as proposed in [37, 16], focus on syntactical aspects and do not consider model checking. To the best of our knowledge, fSMV is the first high-level language for behavioural modelling of SPLs that has efficient model checking algorithms. The main difference between the original work by Plath and Ryan [32] and ours is that we focus on modelling and verifying multiple products at once, while their approach consisted in checking feature combinations exhaustively (limited to pairs of features). Both approaches can be used to detect feature interactions. An advantage of ours is that it is complete and that it produces a propositional formula for each property characterising the interacting features. With [32], an additional aggregation step is required to obtain this information.

Other. For safety analysis of SPLs, Liu *et al.* [28] use Statecharts to model (parts of) reusable components. 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.

Morin *et al.* propose a method to check for inconsistencies between features in adaptive systems [30]. 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 addition to the above, there is a body of related research in the field of feature interaction detection [8]. 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.

8. CONCLUSION

Model checking SPLs is hard because of the exponential number of products to be checked. This adds to the more common phenomenon of state space explosion that is incurred when the individual systems are verified. Our earlier algorithms only address the former kind of blowup, but still face a state explosion problem as they enumerate and visit system states one by one.

In this paper, we proposed symbolic algorithms to tackle this problem. Our contributions are (i) a study of the foundations for symbolic SPL model checking with the introduction of fCTL, (ii) a symbolic algorithm to model check fCTL over FTS, (iii) a usable language and tools based on NuSMV, and (iv) theoretical as well as empirical evaluations of the approach. Benchmarks show that our algorithms cab be considerably faster than standard algorithms.

Although the originality of our work lies in our quest for *SPL-specific* methods and tools, we think that our results partly extend to a larger class of adaptable systems like parametrised components, context-aware systems, etc. This is one of many possible paths for future investigations. Others include mapping visual modelling languages (e.g. Statecharts) to FTS, further evaluation and industrial applications, and further optimisations.

Acknowledgements
We thank Marco Roveri (FBK, Trento), Nicolas Maquet and Jean-François Raskin (ULB, Brussels) for their help with extending NuSMV. Work funded by the FNRS, the Interuniversity Attraction Poles Programme of the Belgian State, Belgian Science Policy (MoVES project) and the BNB.

9. REFERENCES

[1] `http://www.info.fundp.ac.be/~acs/fts`, 2010.

[2] P. Asirelli, M. H. ter Beek, A. Fantechi, and S. Gnesi. A logical framework to deal with variability. In *8th IFM*, number 6396 in LNCS, pages 43–58. Springer, 2010.

[3] P. Asirelli, M. H. ter Beek, S. Gnesi, and A. Fantechi. A deontic logical framework for modelling product families. In *VaMoS'10*, pages 37–44, 2010.

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

[5] D. Batory, D. Benavides, and A. Ruiz-Cortes. Automated analysis of feature models: Challenges ahead. *Comm. ACM*, 49(12):45–47, 2006.

[6] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[7] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. Comp.*, 98(2):142–170, 1992.

[8] 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.

[9] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, volume 131 of *LNCS*, pages 52–71. Springer, 1981.

[10] A. Classen. CTL model checking for software product lines in NuSMV. Technical Report P-CS-TR SPLMC-00000002, University of Namur, 2010.

[11] A. Classen. Modelling with FTS: a collection of illustrative examples. Technical Report P-CS-TR SPLMC-00000001, University of Namur, 2010.

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

[13] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *ICSE 32*, pages 335–344. IEEE, 2010.

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

[15] P. C. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

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

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

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

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

[20] 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.

[21] N. Francez and I. Forman. Superimposition for inter- acting processes. In *Concur'90*, pages 230–245, 1990.

[22] 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.

[23] 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.

[24] K. G. Larsen. Modal specifications. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 232–246. Springer, 1989.

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

[26] K. Lauenroth, S. Töhning, and K. Pohl. Model checking of domain artifacts in product line engineering. In *IEEE/ACM ASE*, pages 269–280, 2009.

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

[28] 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.

[29] K. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

[30] B. Morin, O. Barais, G. Nain, and J.-M. Jézéquel. Taming dynamically adaptive systems using models and aspects. In *ICSE 31*, pages 122–132. IEEE, 2009.

[31] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

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

[33] A. Pnueli. The temporal logic of programs. In *Proc. 18th FOCS*, pages 46–57, 1977.

[34] H. Post and C. Sinz. Configuration lifting: Verification meets software configuration. In *ASE'08*, pages 188–197. IEEE, 2008.

[35] 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.

[36] T. T. Tun, Q. Boucher, A. Classen, A. Hubaux, and P. Heymans. Relating requirements and feature configurations: A systematic approach. In *SPLC'09*, pages 201–210. SEI, CMU, 2009.

[37] 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.