CTL Model Checking for Software Product Lines in NuSMV

Classen, Andreas

Publication date:
2010

Document Version
Early version, also known as pre-print

PReCISE – FUNDP
University of Namur
Rue Grandgagnage, 21
B-5000 Namur
Belgium

TECHNICAL REPORT August 23, 2010 (revised in July 2011)

AUTHORS A. Classen
APPROVED BY P. Heymans
EMAILS {acs}@info.fundp.ac.be
STATUS Draft.
REFERENCE P-CS-TR SPLMC-00000002
PROJECT MoVES
FUNDING FNRS, the Walloon Region, Interuniversity Attraction Poles Programme of the Belgian State of Belgian Science Policy

CTL Model Checking for Software Product Lines in NuSMV

CTL Model Checking for Software Product Lines in NuSMV

Andreas Classen*
PReCISE Research Centre,
Faculty of Computer Science,
University of Namur
5000 Namur, Belgium
acs@info.fundp.ac.be

1 Introduction

In *Software Product Line Engineering (SPLE)*, systems are developed in families [8] and differences between members of a family are generally represented by *features*. A member of a family (called "product") is then defined as a set of features and a *Feature Diagram (FD)* [10, 14] used to concisely model which sets of features are valid products in the *Software Product Line (SPL)*. The model checking problem in this context is more difficult than in single systems engineering as there are $O(2^n)$ different products (n being the number of features) to model and model check [7].

We addressed this problem in [6, 4] with the introduction of *Featured Transition Systems (FTS)*, a formalism to express the behaviour of all products of the SPL in one model. FTS are *Transition Systems (TS)* [1, 3] in which transitions are labelled with the features of an FD (in addition to being labelled with actions). In FTS modelling, features are thus treated as first-class abstractions, which means that the variability concern can be cleanly separated from the behaviour concern. Still, both concerns are linked and this link is exploited in model checking algorithms.

FTS model checking algorithms try to avoid an exponential number of verifications by exploring the FTS rather than the TS of each individual product. We have proposed FTS model checking algorithms for LTL [6] and CTL [5]. Those for LTL were implemented as part of a Haskell library available at the FTS website.[1] This report describes a tool chain for CTL model checking of FTS with the symbolic NuSMV[2] model checker. The tool chain is also available at the aforementioned FTS website.

*FNRS Research Fellow.

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

[2] http://nusmv.irst.itc.it/

2 Background

Let us start with formal definitions of the fundamental concepts used here. Just as in single systems development, the behaviour of an individual product is represented with a TS [1]. 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.

An FTS is basically a TS with another labelling function that labels transitions with features. In [5] we generalised FTS by allowing transitions to be labeled with arbitrary Boolean functions over the features. This slightly more general version of FTS is called FTS$^+$.

Definition 2 (FTS$^+$)**.** *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,*
- *$\gamma : trans \to \left(\{0,1\}^{|N|} \to \{0,1\}\right)$ is a total function, labelling each transition with a* feature expression*, i.e., a Boolean function over the set of features.*

The TS of a single product can be obtained from an FTS$^+$ through *projection*. Projection to some product basically consists in removing all the transitions whose Boolean function is not satisfied by the product.

Definition 3 (Projection in FTS$^+$)**.** *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 $trans' = \{t \in trans | \gamma(t)(f_1 \in p, .., f_n \in p) = 1\}$.*

The semantics of the FTS$^+$ is given by the behaviours of all valid products. Formally, that is the union of the standard TS semantics of all possible projections. This corresponds to the intuition that it represents the behaviour of all products of the SPL.

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

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

Where $[\![d]\!]_{FD} \subseteq \mathcal{P}(N)$ *denotes the* semantics *of the FD* d*.*

For a more gentle introduction to FTS, the interested reader is referred to [6]. A collection of illustrative examples is presented in [4]. A thorough discussion of FTS$^+$ and their relation of basic FTS is provided in [5].

3 Input Language

The input language of our implementation is based on an earlier feature-oriented extension of the NuSMV input language of by Plath and Ryan [12].[3] In this section, we first provide a brief overview of the language. We then show that its models are in fact FTS$^+$ and that it can hence serve as a high-level language for FTS$^+$.

3.1 Language overview

Essentially, a NuSMV model consists of a set of variable declarations and a set of assignments. The variable declarations define the state space and the assignments define the transition relation. In each assignment, the value of a variable in the next state is defined in function of the variable values in the present state. For each variable, there can also be an assignment that defines its initial value. Modules can be used to encapsulate and factor out recurring sub-models. For the purpose of this discussion, we abstract away from them.

The typical example of a NuSMV model (taken from the NuSMV tutorial [2]) is the following.

```
MODULE main
VAR
  request: boolean;
  state:   {ready, busy};
ASSIGN
  init(state) := ready;
  next(state) := case state = ready & request = 1: busy;
                      1: {ready, busy};
                 esac;
```

[3] More precisely, they used the earlier SMV model checker. The input language of NuSMV is almost identical.

The above module describes a controller that is either busy treating a request or ready to receive one. Requests are controlled by the environment and modelled as a non-deterministic variable. When there is a request and the controller is ready, it will treat the request and be busy, it may continue to be busy for a while and return to ready once the request is treated.

Let us formalise these concepts in order to facilitate the definition of the fSMV/FTS$^+$ translation of the next section.

Definition 5. *Let V be a set of variables, D a set of (finite) domains or types, and $E(V)$ the set of all SMV expressions over V. Let $A(V)$ be the set of assignments $A(V) \subseteq \{\texttt{-}, \texttt{init}, \texttt{next}\} \times V \times E(V)$, that is, a set of triples (s, d, e) where s distinguishes between `d` (the `-`), `init(d)` or `next(d)` for $d \in V$, and $e \in E$ is an expression.*[4] *A base system m is a tuple $m = (v, \tau, a, p)$, where*

- *$v \subseteq V$ is a set of variables,*
- *$\tau : V \to D$ a function assigning a domain to each variable,*
- *$a \subseteq A(v)$ is a set of assignments, and*
- *$p \subseteq \mathcal{P}(v) \times \mathcal{P}(v)$ is a (possibly empty) set of processes. A process is a couple (v_p, w_p) where v_p denotes the set of variables read by the process and $w_p \subseteq v_p$ denotes the set of variables written by the process. Furthermore, SMV requires that the sets of written variables do not overlap.*

For a model without parallel composition, the set p is empty. The semantics of a base system is a TS or the parallel composition of several TSs [11].

The language by Plath and Ryan [12], hereafter called fSMV, is based on *superimposition* [9]. A feature basically describes the changes to be made to the original system. There are three categories of changes a feature can make:

`INTRODUCE` new variables into the system.

`IMPOSE` a new definition of an existing variable. This means that in the presence of the feature, the `init` or `next` state definition of the variable will be replaced. An `IMPOSE` clause can be guarded, meaning that it only has an effect if a certain condition holds.

`TREAT` existing variables differently. When the value of the variable is read, e.g. inside the definition of some other variable, the value read is modified by the feature. A `TREAT` clause can also be guarded, but this is just syntactic sugar [12] and will be omitted in our discussion.

Formally, an fSMV model is defined as follows.

Definition 6. *An fSMV model is a pair (b, G), where b is a base model and G an (ordered) list of features. Each feature $f \in G$ is a tuple consisting of*

- *$v_f \subseteq V$, a set of new variables;*
- *$\tau_f : v_f \to D$, a type function;*

[4] An expression alone is not an assignment, it just defines a value.

- $p_f : p \to \mathcal{P}(v_f) \times \mathcal{P}(v_f)$, *a function that tells for each process whether the new variables belong to it (read and write respectively); sets of written variables cannot overlap;*
- $a_f \subseteq A(v \cup v_f)$, *a set of* `INTRODUCE` *assignments;*
- $m_f \subseteq E(v \cup v_f) \times A(v)$, *a set of guarded* `IMPOSE` *assignments (for* $(e, a) \in m_f$, *e is the guard); and*
- $t_f \subseteq A(v)$, *a set of* `TREAT` *assignments.*

We intentionally keep these definitions at a high level of abstraction. They are sufficiently detailed to make the following discussion precise and abstract enough to make it intuitive. In particular, we do not detail the syntax or semantics of expressions or of the types. Furthermore there are a number of rules on what constitutes a valid fSMV model (wrt. types, variable names, etc.) which we also omit. The interested reader is referred to [11, 13] for a detailed formal definition of SMV, NuSMV and fSMV.

As an example, consider the feature *sleep* which adds a switch to the system that causes it to not accept any further request. The switch is modelled with a new non-deterministic variable `sleep`. The system is changed in such a way that if the system is sleeping and finished treating requests, then it will stay ready, not accepting any new requests.

```
FEATURE sleep
INTRODUCE
   VAR sleep: boolean;
CHANGE
   IF sleep = 1 & busy = 0 THEN IMPOSE next(state) := ready;
```

Composing a base system and a feature yields a new base system. The composition of base system and the preceding *sleep* feature gives the following system.

```
MODULE main
VAR
  request: boolean;
  state:   {ready, busy};
  sleep:   boolean;
ASSIGN
  init(state) := ready;
  next(state) := case sleep = 1 & busy = 0: ready;
                      1: case state = ready & request = 1: busy;
                              1: {ready, busy};
                         esac;
                 esac;
```

Composition can be formally defined as follows.

Definition 7. *Composition of a base system* $b = (v,\ \tau,\ a,\ p)$ *and a feature* $f = (v_f, \tau_f, p_f, a_f, m_f, t_f)$ *is noted* $b \otimes f$ *and produces a new base system* $b' = (v', \tau', a', p')$*, where*

- $v' = v \cup v_f$ *and* $\tau' = \tau \cup \tau_f$
- a' *is obtained by first applying* t_f *to* a*, then* m_f*, and finally adding* a_f*, formally:*

$$\begin{aligned} a' &= a_m \cup a_f \\ a_m &= \{(s,d,e') | (s,d,e) \in a_t \wedge \exists (g,(s',d',e'')) \in m_f \\ &\qquad \bullet\ s = s' \wedge d = d' \\ &\qquad\qquad \Rightarrow e' = \texttt{case}\ g : e''; 1 : e\ \texttt{esac;}\} \\ a_t &= \{(s,d,e') | (s,d,e) \in a \\ &\qquad\qquad \wedge\ e' = e[s' \leftarrow e'' | (s',e'') \in t_f]\} \end{aligned}$$

  where $e[s_1 \leftarrow e_1, \ldots, s_n \leftarrow e_n]$ *denotes the simultaneous replacement of all* s_i *in* e *by* e_i*,* $1 \leq i \leq n$*.*
- $p' = \{(v \cup v_f, w \cup w_f) | (v,w) \in p \wedge p_f(v,w) = (v_f, w_f)\}$

3.2 From fSMV to FTS$^+$

Composition produces a new NuSMV model that has no information about the features it contains. Hence, to perform verification, products have to be composed and model checked individually, which leads to an enumerative approach. This is exactly what FTS$^+$ model checking intends to avoid. Alternatively, an fSMV model can be interpreted as an FTS$^+$, so that FTS$^+$ model checking is equivalent to checking all compositions of the fSMV model.

We first recap the concept of transition in SMV. The set of assignments of an SMV model can be interpreted as a Boolean function with two parameters: a valuation of all variables in one state and in the next state. Given two states $s, s' \in S$, there is a transition $s \rightarrow s'$ if and only if the variable values in these states satisfy the Boolean function induced by the set of assignments a. We write this $s, s' \models a$. Similarly, the set of *init* assignments (noted $i_a = \{(\texttt{init}, var, e) \in a\}$) can be interpreted as a Boolean function with one such parameter. A state s satisfying all init assignments is written $s \models i_a$. We assume that the condition IV of [12] is verified: the order of features is irrelevant, i.e. two features commute if both orders are well-defined: $f_i \otimes f_j = f_j \otimes f_i$. This assumption allows us to consider a product as a set of features, instead of a list. We have the following result.

Theorem 8. *For each commutative fSMV model without parallel composition, there is an FTS*$^+$ *whose behaviours are those of all products in the fSMV model.*

Proof. Given an fSMV model $(b, \{f_1, \ldots, f_n\})$ with $b = (v, \tau, a, \emptyset)$ and $f_i = (v_i, \tau_i, p_i, a_i, m_i, t_i)$, construct an FTS$^+$ $(S, Act, trans, init, AP, L, d, \gamma)$ where

(8.1) *init* is a designated initial state $init = s_0$;

(8.2) The set of states is the set of all variable values, plus the fresh initial state: $S = \{s_0\} \cup \prod_{d \in v} \tau(d) \times \prod_{i \in \{1,n\}} \prod_{d \in v_i} \tau_i(d)$;

(8.3) Action labels are not used, transitions have a dummy label: $Act = \{\epsilon\}$.

(8.4) Atomic propositions are derived from the states;

(8.5) The set of products can be derived from the set of valid feature lists of the fSMV model: each required element (module or variable) has to be introduced by a previous feature; and an element cannot be introduced twice;

(8.6) Let p be a set of features, and l one of the lists from which it was derived (see above). Since all such lists give the same results by commutativity, $b \otimes p$ can be defined as $b \otimes l$. Let a_p be the transition relation of $b \otimes p$. For correctness, we must have: $(s, s') \models a_p$ iff $\gamma(s, s')(p) = 1$. Therefore we use this as the definition of γ.

Similarly, we consider the initial states as resulting from an assignment at time 0, and thus as a transition from the fresh initial state $s_0 : \gamma(s_0, s')(p) = s' \models i_p$, where i_p is the condition on the initial state given by the `init` assignments after application of the features in p.

The obtained FTS$^+$ has *by construction* the same behaviours as the fSMV, all prefixed by a single new transition. This transition is due to the new initial state that has to be added because a feature in fSMV can modify the initial state. □

For fSMV models whose base system is the parallel composition of several processes, we propose the following result.

Theorem 9. *For each fSMV model with parallel composition, there exists a set of FTS$^+$ whose parallel composition has a set of behaviours equal to those of all products in the fSMV model.*

Proof. Given an fSMV model $(b,\ fx)$, with $b = (v, \tau, a, \{p_1 \dots p_k\})$, construct the FTS$^+$ of each process $p \in \{p_1 \dots p_k\}$, with $p = (v, w)$, as follows

- Create a base system $b_p = (v_p, \tau, a_p, \emptyset)$ where
  - $v_p = v$, the variables are those of the process
  - $a_p = \{m \mid m \in a \wedge m \in A(v)\}$, the assignments are those that define variables of the process
- Transform the fSMV model (b_p, fx) into an FTS$^+$ fts_p as described in the proof of Theorem 8.
- The process context of fts_p is $c_p = (v, w)$.

The resulting FTS is given by $||_{p \in \{p_1 \dots p_k\}} fts_p, c_p$ □

Theorem 9 basically states that the fSMV language is a subset of the FTS$^+$ language. The following theorem establishes that the converse also holds, i.e., that both languages are expressively equivalent.

Theorem 10. *Any FTS$^+$ can be translated into an fSMV.*

Proof. Given an FTS$^+$ (S, Act, $trans$, $init$, AP, L, d, γ), construct an fSMV model (b, F) with $b = (v, \tau, a, \emptyset)$, where

(10.1) One variable of the fSMV, *state*, is used to encode all the states of the FTS: $\tau(state) = S$. Let $d = (N, px)$, for every feature $f \in N$, the fSMV will have a variable f with $\tau(f) = \{0, 1\}$. Hence, $v = \{state\} \cup N$;

(10.2) The initial value of the feature variables is 0, which remains constant. The assignments related to the feature variables are thus $a_F = \{(\texttt{-}, f, 0) | f \in N\}$. The initial value of the *state* variable is the initial state of the FTS$^+$, and its `next` value is derived from the transition relation of the FTS$^+$. The assignments related to the *state* variable are

$$a_s = \{(\texttt{init}, state, init), (\texttt{next}, state, \texttt{case}\ case_1 \ldots case_k\ \texttt{esac;})\}$$

where the $case_i$ are given by $\{state = s\ \texttt{\&}\ \gamma(s \xrightarrow{\alpha} s')\texttt{:}\ s'\texttt{;} \mid s \xrightarrow{\alpha} s' \in trans\}$. The set of assignments is then $a = a_F \cup a_s$;

(10.3) Each feature imposes that its associated variable (which is part of the base system) takes the value 1, i.e., $F = \{(\emptyset, \emptyset, \emptyset, \emptyset, \{(1, (\texttt{-}, f, 1))\}, \emptyset) | f \in N\}$.

In consequence, composition of a base system with a set of features yields a TS of which all transitions whose $\gamma(s \xrightarrow{\alpha} s')$ evaluates to 0 for the feature variables have been removed. This corresponds exactly to projection as defined in Definition 3. In order to limit the set of valid feature combinations to those of $[\![d]\!]$, an additional constraint could be added to each $case_i$ in the definition of the *state* variable: the Boolean function equivalent of the set $[\![d]\!]$. (Another solution would be to use the `IVAR` construct of NuSMV to add the Boolean function equivalent of $[\![d]\!]$ as an invariant to the model.) □

Note that Theorem 10 does not have to take parallel composition into account directly. Any parallel composition of two FTS$^+$ is an FTS$^+$ itself, and can hence be translated into an fSMV. Theorem 10 also does not need process contexts, meaning that it can be used for FTS$^+$ without process contexts, too.

4 FTS$^+$ model checking in NuSMV

As input, our implementation of FTS$^+$ model checking in NuSMV uses an fSMV model (i.e. a base system and a list of features) as well as one or more CTL properties.[5] For each property it determines the products for which it is satisfied—without resorting to an enumerative approach that model checks all possible products individually.

The NuSMV extension and all other scripts discussed here are available at the FTS website.[6]

[5] Properties expressed with the fCTL logic discussed in [5] can be used, too.

[6] `http://www.info.fundp.ac.be/~acs/fts`

4.1 Composition

To achieve this, features have to be encoded as part of the model. In order to be able to reuse as much of the existing NuSMV machinery as possible, we decided to encode features as part of the states. This is a slight derivation from the symbolic encoding proposed in [5], where features are (as in pure FTS$^+$) encoded as part of the transition relation. Basically, each feature becomes a Boolean state variable, that is non-deterministically initialised and whose value never changes. Every change performed by a feature is guarded (at composition time) by the corresponding feature variable.

All this is encapsulated by the composition operator. It differs from the one discussed in the previous section in that the resulting NuSMV model has information about which features it contains. For the example of the previous section, the composition would result in the following NuSMV model.

```
MODULE features
VAR
  fSleep: boolean;
ASSIGN
  init(fSleep) := {0,1};
  next(fSleep) := fSleep;

MODULE main
VAR
  f:       features;
  request: boolean;
  state:   {ready, busy};
  sleep:   boolean;
ASSIGN
  init(state) := ready;
  next(state) := case f.fSleep = 1 & sleep = 1 & busy = 0: ready;
                      1: case state = ready & request = 1: busy;
                              1: {ready, busy};
                         esac;
                 esac;
```

First, a module containing all features (in this case, a single one) is added to the system. To each feature corresponds one variable in this module, the variable name being the feature name (with the first letter in uppercase) prefixed by the letter `f`. The feature module is called `features` and is used in the main module as a variable named `f`. A parameter called `f` is added to all other modules in the model (not shown in the previous example), so that the feature module is accessible inside the whole model. The feature variables can thus be referenced in all modules.

These naming conventions allow us to easily distinguish feature variables from 'normal' variables. All feature variables have the prefix `f.f`: the first `f` identifies the variable of the main module that holds the feature module and

the second `f` is the one prefixed to every feature variable. We need to be able to distinguish feature variables from the other variables when calculating the products for which a certain property holds. An alternative to the naming convention would have been to extend the NuSMV input language. We chose a naming convention as this necessitates far less changes to the NuSMV codebase.

This composition operator is implemented in a PHP script `compose.php`. The script reads the base system from standard input, takes the path to a feature file in parameter, and writes a new NuSMV model to standard output. The output can be piped to another composition call with a different feature. The script implements the 'basic' composition as specified in Definition 7 as well as our method described above, activated with the command line switch `-l`. For the running example, the command line might look as follows.

```
php compose.php -l sleep.feat < base.smv > baseWithSleep.smv
```

4.2 Model checking

The output of the composition tool is a normal NuSMV model and can be model checked directly by NuSMV. However, standard NuSMV model checking does not fully exploit the feature encoding given in the previous section. Since NuSMV executes the standard CTL model checking algorithm, it will report $false$ if it finds a counterexample. More precisely, it will return $false$ if just one of the products violates the property.

Basically, given a property ϕ, the algorithm will compute a Boolean function $\chi_{Sat(\phi)}(\overline{s}, \overline{p})$, where $\overline{s}$ (resp. $\overline{p}$) is the Boolean encoding of some state (resp. some product). $\chi_{Sat(\phi)}$ is true for all states and products that satisfy the property. The normal model checking algorithm will just check whether there exists some initial state for which $\chi_{Sat(\phi)}(\overline{s}, \overline{p})$ is $false$. Unable to distinguish between feature variables (belonging to $\overline{p}$) and normal variables (belonging to $\overline{s}$), the test will existentially quantify over the feature variables which corresponds to considering a single product only.

As discussed in [5], there is sufficient information to determine exactly which products violate and which satisfy the property. The idea is to only quantify $\chi_{Sat(\phi)}$ existentially over the variables that do not represent features. The result is a Boolean function over the feature variables that represents exactly the products for which the property holds.

This is what is done by our NuSMV extension. It adds a command line switch `-fbdd` that, if set, performs the described calculation and prints the boolean function into the normal NuSMV output. The extension itself is provided as a patch for NuSMV 2.5.0.

5 Benchmarks

For benchmarking we used the elevator system by Plath and Ryan [12]. We extended the SMV models provided with the original paper in two ways. First, we made the number of floors (initially fixed at five) variable. For this, we had

to extended the NuSMV input syntax with quantifiers. These quantifiers are implemented by a preprocessor in form of a PHP script. Secondly, we added four more features to the system, giving a total of nine features. All features are independent, which means that there are 2^9 products.

5.1 Elevator System

The elevator system is comprised of a number of platform buttons and a number of cabin buttons. There is a single button on each platform, which calls the elevator. The button press is modelled non-deterministically, and a pressed button remains pressed until the elevator has served the floor and its doors opened. The elevator will always serve all requests in its current direction before it stops and changes direction. When serving a floor, the lift doors open and close again. There are nine features that modify the behaviour of the lift. Those marked with an asterisk were added by us.

Antiprank.* The lift buttons will not remain pushed until served. They have to be held pushed by a person.

Empty. If the lift is empty, then all requests made in the cabin will be ignored.

Executive floor. One floor of the building has priority over the other floors and will be served first.

Open if idle.* When idle, the lift opens its doors.

Overload The lift will refuse to close its doors when it is overloaded.

Park. When idle, the lift returns to the first floor.

Quick close.* The lift door cannot be kept open by holding the platform button pushed.

Shuttle.* The lift will only change direction at the first and last floor.

Two-thirds full. When the lift is two-thirds full, it will serve cabin calls before platform calls.

To test the correctness of our approach we reduced the example to the five features from [12] and managed to reproduce the feature interactions reported in the original paper. Subsequently, we made some minor modifications to the model to accommodate the additional features.

5.2 Methodology and Results

We ran a number of benchmarks on the elevator system, using properties of the base system shown in Table 1. Each property was benchmarked individually. The property numbers reported in the statistics refer to the numbers in Table 1, also given to the properties 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 reported benchmarks compare (for each property)

Table 1: Benchmarked properties

ID	Property
01	`AG (landingBut2.pressed -> AF (lift.floor=2 & lift.door=open))`
01'	`!AG (landingBut2.pressed -> AF (lift.floor=2 & lift.door=open & lift.direction=down))`
02	`AG (liftBut3.pressed -> AF (floor=3 & door=open))`
03a	`AG (floor=2 & liftBut6.pressed & direction=up -> A[direction=up U floor=6])`
03b	`AG (floor=6 & liftBut1.pressed & direction=down -> A[direction=down U floor=1])`
04	`!AG (door=closed -> AF door=open)`
05a	`EF(floor=1 & idle & door=closed & AX(door=closed))`
05b	`AG (floor=1 & idle & door=closed & AX(door=closed) -> EG (floor=1 & door=closed))`
05-part	`EF(AX(door=closed))`
05c	`EF(floor=3 & idle & door=closed & AX(door=closed))`
05d	`AG (floor=3 & idle & door=closed & AX(door=closed) -> EG (floor=3 & door=closed))`
05e	`EF ( EG ( door = closed ))`
05'	`!AG(floor=4 & idle -> E [idle U floor=1])`
06	`!AG ((floor=3 & !liftBut3.pressed & direction=up) -> door=closed)`
07	`!AG ((floor=3 & !liftBut3.pressed & direction=down) -> door=closed)`

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

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 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 at the FTS website.

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,

Table 2: Benchmark results for the elevator system with four floors.

Property	**Value**	**Enumerative**	**Single**	**Speedup**
01	$false$	17.84	0.14	127.43
01'	$true$	15.37	0.05	307.40
04	$false$	18.19	1.06	17.16
02	$false$	19.23	0.22	87.41
03a	$false$	20.48	1.84	11.13
03b	$false$	21.23	1.76	12.06
05a	$false$	20.09	3.23	6.22
05b	$true$	14.36	0.03	478.67
05-part	$true$	16.47	0.06	274.50
05c	$false$	19.94	1.86	10.72
05d	$true$	14.68	0.03	489.33
05e	$false$	18.3	1.06	17.26
05'	$false$	19.89	1.62	12.28
06	$true$	18.89	1.2	15.74
07	$true$	19.27	2.57	7.50

Table 3: Benchmark results for the elevator system with five floors.

Property	**Value**	**Enumerative**	**Single**	**Speedup**
01	$false$	29.38	0.44	66.77
01'	$true$	24.76	0.09	275.11
04	$false$	34.02	4.62	7.36
02	$false$	33.16	0.82	40.44
03a	$false$	37.98	6.3	6.03
03b	$false$	39.43	6.32	6.24
05a	$false$	39.77	13.99	2.84
05b	$true$	22.7	0.03	756.67
05-part	$true$	29.25	0.16	182.81
05c	$false$	35.52	8.66	4.10
05d	$true$	23.44	0.04	586.00
05e	$false$	34.09	4.63	7.36
05'	$false$	40.21	8.14	4.94
06	$true$	34.55	4.56	7.58
07	$true$	35.9	7.57	4.74

Table 4: Benchmark results for the elevator system with six floors.

Property	**Value**	**Enumerative**	**Single**	**Speedup**
01	$false$	44	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

Table 5: Benchmark results for the elevator system with seven floors.

Property	**Value**	**Enumerative**	**Single**	**Speedup**
01	$false$	66.89	3.45	19.39
01'	$true$	44.34	0.17	260.82
04	$false$	214.75	109.67	1.96
02	$false$	86.98	5.58	15.59
03a	$false$	160.43	51.35	3.12
03b	$false$	169.91	66.45	2.56
05a	$false$	487.98	571.69	0.85
05b	$true$	38.39	0.04	959.75
05-part	$true$	114.38	0.55	207.96
05c	$false$	269.19	257.98	1.04
05d	$true$	38.62	0.06	643.67
05e	$false$	214.13	112.79	1.90
05'	$false$	568.56	241.53	2.35
06	$true$	142.42	48.37	2.94
07	$true$	160.3	128.84	1.24

Table 6: Benchmark results for the elevator system with eight floors.

Property	**Value**	**Enumerative**	**Single**	**Speedup**
01	*false*	99.14	4.96	19.99
01'	*true*	62.71	0.15	418.07
04	*false*	337.47	414.32	0.81
02	*false*	139.58	6.06	23.03
03a	*false*	312.05	57.65	5.41
03b	*false*	332.49	81.35	4.09
05a	*false*	2180.58	2232.39	0.98
05b	*true*	51.26	0.04	1281.50
05-part	*true*	211.63	0.48	440.90
05c	*false*	851.58	899.2	0.95
05d	*true*	52.27	0.07	746.71
05e	*false*	337.81	407.84	0.83
05'	*false*	2441.67	887.8	2.75
06	*true*	263.68	102.39	2.58
07	*true*	325.31	439.25	0.74

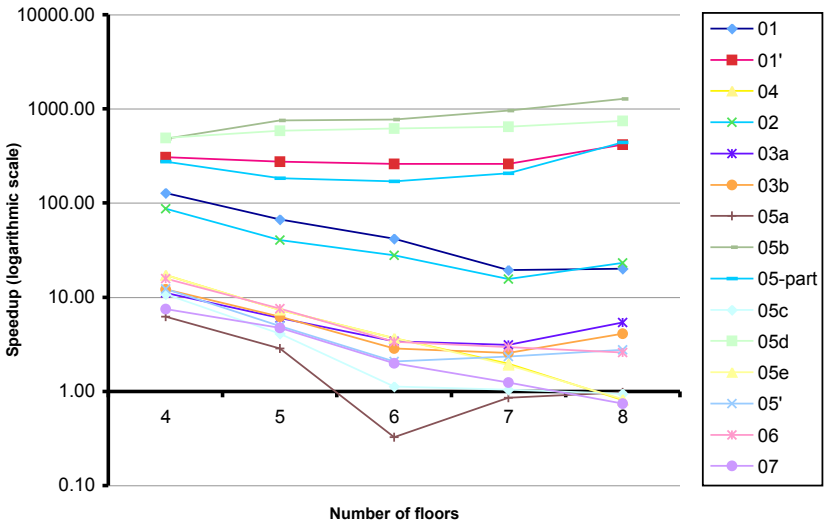

Figure 1: Evolution of speedup with the number of floors (logarithmic scale).

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 for each level. 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 for the *enumerative* case.

Results of these benchmarks are shown in Tables 2, 3, 4, 5 and 6.

The results show that our approach achieves order-of-magnitude speedups over the enumerative approach. More precisely, we observed that our approach is on average 130 times faster than the enumerative one. These observations are reported for each property in Figure 1, 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 levels, the outliers tend to become more extreme (the high speedups grow, the low speedups descend). We believe that this reflects the importance of the static variable ordering for large models.

In order to limit bias, we went to great lengths to ensure that the *enumerative* benchmarks were as efficient as possible. 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).

Acknowledgements

We are grateful to Marco Roveri from FBK (Trento) who was of great help for implementing the NuSMV extensions, and who helped us with static variable orderings and seemingly arbitrary runtimes of NuSMV. Thanks also go to Nicolas Maquet and Jean-François Raskin from ULB (Brussels) who got us started with the NuSMV hacking in the initial version of the tool.

References

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

[2] R. Cavada, A. Cimatti, G. Keighren, E. Olivetti, M. Pistore, and M. Roveri. *NuSMV 2.2 Tutorial*.

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

[4] A. Classen. Modelling with FTS: a collection of illustrative examples. Technical Report P-CS-TR SPLMC-00000001, PReCISE Research Center, University of Namur, 2010. Available online.

[5] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic model checking of software product lines. Submitted for review to the International Conference on Software Engineering, August 2010.

[6] 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. ACM, 2010. Acceptance rate: 13.7

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

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

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

[10] 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.

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

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

[13] M. Plath and M. D. Ryan. The feature construct for smv: Semantics. In *FIW VI*, pages 129–144. IOS Press, 2000.

[14] 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.