Model Checking for Software Product Lines with SNIP

Classen, Andreas; Cordy, Maxime; Heymans, Patrick; Legay, Axel; Schobbens, Pierre-Yves

Publication date:
2012

Document Version
Early version, also known as pre-print

Int J Softw Tools Technol Transfer (2012) 14:589–612
DOI 10.1007/s10009-012-0234-1

MTM

Model checking software product lines with SNIP

Andreas Classen · Maxime Cordy · Patrick Heymans · Axel Legay · Pierre-Yves Schobbens

Published online: 14 June 2012

Abstract We present SNIP, an efficient model checker for software product lines (SPLs). Variability in software product lines is generally expressed in terms of *features*, and the number of potential products is exponential in the number of features. Whereas classical model checkers are only capable of checking properties against each individual product in the product line, SNIP exploits specifically designed algorithms to check all products in a single step. This is done by using a concise mathematical structure for product line behaviour, that exploits similarities and represents the behaviour of all products in a compact manner. Specification of an SPL in SNIP relies on the combination of two specification languages: TVL to describe the variability in the product line, and fPromela to describe the behaviour of the individual products. SNIP is thus one of the first tools equipped with specification languages to formally express both the variability and the behaviours of the products of the product line. The paper assesses SNIP and suggests that this is the first model checker for SPLs that can be used outside the academic arena.

Keywords Model checking · Product lines · Tool · Language · Feature

A. Classen and M. Cordy are FNRS research fellows..

A. Classen (✉) · M. Cordy (✉) · P. Heymans · P.-Y. Schobbens
University of Namur, Namur, Belgium
e-mail: acs@info.fundp.ac.be

M. Cordy
e-mail: mcr@info.fundp.ac.be

A. Legay
IRISA/INRIA Rennes, Rennes, France

A. Legay
University of Liège, Liège, Belgium

P. Heymans
INRIA Lille-Nord Europe, Université Lille 1, LIFL, CNRS, Lille, France

1 Introduction

Software product line (SPL) engineering is an increasingly popular software development paradigm for building families of similar software products. Many of those software products are used in critical applications such as automotive or avionics. This requires a solid evidence that they indeed work correctly with respect to their requirements and intended properties. For example, a desired property for the file transfer protocol studied in Sect. 6 is: "*A sent message eventually reaches its destination*".

A simple but cumbersome approach for product line verification consists in applying classical model checking algorithms [38] on each individual product of the family. However, for an SPL with n features, this would lead to 2^n calls of the model checking algorithm. This solution is clearly unsatisfactory and should be replaced by new approaches that take the variability within the family into account. Those approaches often rely on compact mathematical representations on which a specialized model checking algorithm can be applied. The main difficulties are (1) to develop such a model checking algorithm, and (2) to propose mathematical structures that are compact and flexible enough to take the variability of the family and its specification into account.

In [13], we introduced *featured transition systems* (FTSs), an extension of transition systems used to represent the behaviour of all the products of an SPL in a single compact structure. We also proposed new model checking algorithms that make use of the compact structure of FTSs to verify the whole SPL in a single execution. More precisely, these

semi-symbolic algorithms model-check the SPL against temporal properties expressed in *linear temporal logic* (LTL) [35]. Those algorithms, capable of identifying all the products of the SPL that do satisfy a property, are called *FTS algorithms*.

This article presents SNIP, an SPL model checking tool that implements our theory. Concretely, SNIP implements the FTS algorithms for verifying SPLs against LTL properties. Not only does SNIP put these theoretical results into practice, it also reflects our concern of combining them with high-level specification languages for product line specification—a mandatory step to transfer our results from theory to practical applications. More precisely, the specification of the SPL in SNIP relies on the combination of two specification languages: TVL [11] to describe the variability in the family, and fPromela to describe the behaviour of the individual products in a compact manner.

fPromela is an extension of Promela, the modelling language used by the well-known model checker SPIN [23]. The syntax of Promela is close to procedural programming languages such as Pascal or C. Its relatively low complexity makes it an easy-to-learn modelling language. fPromela extends Promela by adding a guard operator, which allows to make a transition available only to a subset of the products. TVL is a text-based feature modelling language that allows to declare the features of an SPL, as well as the mathematical relations and constraints between them. Hence, the TVL model defines the set of products of the SPL that are valid with respect to the specified constraints.

Given an LTL property and an SPL described in TVL and fPromela, SNIP uses the FTS algorithms to model check the whole product line instead of each product individually. In case there are products that do not satisfy the property, the tool will identify one (or all) of these product(s) together with a counterexample. To the best of our knowledge, SNIP is the only SPL model checking tool with a high-level specification language that is able to model check SPLs in an efficient manner.

In addition to the FTS algorithms, SNIP also implements a naïve approach in which each product is model checked individually. This permits us to compare the two approaches and assess the efficiency of the FTS algorithms. This is done by measuring the time needed by both algorithms implemented in SNIP to verify the SPL of a file transfer protocol against several properties.

Structure of the paper. Section 2 surveys related work. Section 3 presents the theoretical foundations of SNIP. Section 4 gives the syntax and semantic of fPromela and its relation to FTSs, hence showing soundness and correctness of the implementation. Section 5 describes the user interface of SNIP and discusses its architecture. Finally, Sect. 6 presents the setup and the results of a case study.

2 Brief overview of related work

SNIP is one of the first tools for SPL model checking, which means that there are only few approaches that can be compared to what we propose in this paper. For a thorough discussion of the wider (mostly theoretical) related work, the interested reader is referred to [14].

Let us begin with a brief overview of SPL model checking before we discuss tools related to SNIP. A number of fundamental models for product line behaviour have been proposed in the literature. Most of those proposals do not embed any verification procedure. Among such existing approaches, one finds those based on UML, e.g. [16,28,33,39], which extend existing UML diagram types (sequence diagrams, state machines) to allow them to model SPL behaviour. However, verification can only be performed after a model of a specific product has been derived. This leads to a scalability problem as the number of products is potentially huge.

More formal approaches to modelling SPL behaviour are based on modal transition systems (MTSs) [18,30] and modal I/O automata [31]. In these approaches, transitions can be mandatory (*required* transitions) or optional (*allowed* transitions). As expected, *allowed* transitions can be used to model variability, and an MTS essentially specifies a family of behaviours. In addition to basic MTSs, there are proposals to extend them by introducing variability operators to specify cases in which a specific number of outgoing transitions may be taken [17]. In [2,3], the authors propose a deontic logic interpreted over MTSs, that can be used to express both behavioural properties, and constraints over features. The CCS process algebra was extended in a similar way [21], with an operator that allows to model variability in the form of alternative choice between two processes.

All these approaches are severely limited in that they cannot relate a behaviour of a model to a specific product or feature of the SPL. For example, MTSs are not capable to capture the features that make a transition optional; similarly, choices between processes in [21] are not linked to features. In consequence, a verification algorithm cannot identify bad products (those that violate a property) or interacting features. Our recent work [12–14] as well as [32] overcome this limitation by explicitly linking transitions to features. The key idea of linking transitions to features is transposed as-is to SNIP, the tool we focus on in this paper.

There are a number of tools similar to SNIP. In our earlier work [12], we developed an extension of the NuSMV model checker [8] which can model check SPLs expressed using the fSMV language [34]. This tool uses the fully symbolic FTS algorithm from [12], whereas SNIP uses the semi-symbolic algorithms from [13,14]. SNIP differs in other ways from our NuSMV extension. Its modelling language, fPromela, is based on annotation rather than composition [26].

Moreover, SNIP is integrated with a feature diagram language, TVL [11].

An early approach to SPL model checking was proposed by Post and Sinz [36], with a technique called *lifting*. It consists in incorporating the information about allowed products (the *variability*) into the verifiable model itself. A similar approach is followed by SPLVerifier [1], in which features are modelled as separate and composable units. Both approaches use a classical model checker to detect violations. A problem with these approaches is that they stop once a violating product is found. In such a case, they cannot be used to compute the products that do satisfy the property. This also makes it hard to determine the features responsible for a violation. SNIP's use of our FTS algorithms overcomes these problems.

For a more general overview of the topic of software model checking, the interested reader is referred to [24].

3 Foundations

In this section, we briefly recap the theoretical foundation of SPL verification with FTS algorithms. The interested reader is redirected to [10,14] for more details.

3.1 FTS

In our theory, the behaviours of an individual product are represented with *Transition Systems* (TSs) [4]. A TS is a directed graph whose transitions are labelled with actions, and whose states are labelled with atomic propositions.

Definition 1 A 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 and $L : S \to \mathcal{P}(AP)$ labels each state with the propositions that are true in it.

Where $\mathcal{P}$ denotes the power set.

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 extends the concept of TS to SPLs. Roughly speaking an FTS is nothing more than a TS whose transitions are labeled with the set of products for which the transition is available. An FTS comes together with a *Feature Diagram* (FD) that defines the set of features, and captures the set of valid products in the SPL (see, e.g., [25,37]). FDs can be seen as a constraint language over a set of propositional symbols (the features). An FD can record information such as incompatibility, or dependency between features. For the purpose of this paper, it is sufficient to assume that the semantics of an FD d, denoted $[\![d]\!]_{\text{FD}}$, is its set of valid *products*, i.e., a set of sets of features: $[\![d]\!]_{\text{FD}} \subseteq \mathcal{P}(N)$ where N is the set of features defined in the FD.

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,
- $\gamma : trans \to \left(\{\bot, \top\}^{|N|} \to \{\bot, \top\}\right)$ is a total function, labelling each transition with a *feature expression*, i.e., a Boolean function over the set of features. By $[\![\gamma(t)]\!]$, we denote the set of products that satisfy $\gamma(t)$.

In SNIP, FTSs are specified with fPromela and their corresponding FD is specified in TVL [11]. The interaction between the two languages shall be studied in Sect. 4. Observe that the TS of a particular product of the SPL is obtained by removing all transitions of the FTS whose feature expression evaluates to $false$ in the product. This is called *projection*.

Definition 3 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 $trans' = \{t \in trans \mid p \in [\![\gamma(t)]\!]\}$.

The FTS represents the behaviour of all the products. Its semantics is thus the union over their projections.

Definition 4 $[\![fts]\!]_{FTS} = \bigcup_{c \in [\![d]\!]_{\text{FD}}} [\![fts_{|c}]\!]_{TS}$

3.2 The temporal logic fLTL

A set of features defines a product, which has particular behaviours. However, as is clear from Definition 4, features in an FTS are not part of the behaviours or of the states of the system. This means that existing logics, such as Linear Temporal Logic (LTL) or Computation Tree Logic (CTL) can be used to specify properties over an FTS. LTL consists of standard Boolean connectives such as $\wedge$ and $\neg$, as well as a set of *temporal operators*. These operators are: *next*, $\bigcirc\phi$, which requires that the next state in an execution satisfies ϕ; and *until*, $\phi_1 U \phi_2$, which requires that ϕ_1 holds until ϕ_2 is satisfied (and ϕ_2 has to be satisfied at some point). Other temporal operators can be derived from U: *eventually*, $\Diamond\phi$, defined as $\top U \phi$; and *always*, $\Box\phi$, which is defined as $\neg\Diamond\neg\phi$.

In [13], we propose a slightly modified version of LTL that allows to quantify over the set of products. This logic, which is the one supported by SNIP, is called *feature LTL* (fLTL) and is syntactically defined as follows.[1]

[1] In [12], we proposed a similar extension for CTL. The model checking algorithm of this logic is not part of the SNIP toolset.

Definition 5 An fLTL property ψ is an expression $\psi = [\chi]\phi$ where $\chi : \{\bot, \top\}^{|N|} \to \{\bot, \top\}$ is a feature expression and ϕ an LTL property, i.e.,

$$\phi ::= \top \mid a\ (\in AP) \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \bigcirc \phi \mid \phi_1 U \phi_2.$$

The LTL property is interpreted over infinite executions of a TS, and satisfaction for an execution π is defined the usual way:

$$\begin{aligned}
&\pi \models \top \\
&\pi \models a && \iff a \in L(\pi_0) \\
&\pi \models \neg\phi && \iff \pi \not\models \phi \\
&\pi \models \phi_1 \wedge \phi_2 && \iff \pi \models \phi_1 \text{ and } \pi \models \phi_2 \\
&\pi \models \bigcirc\phi && \iff \pi_{[1\ldots]} \models \phi \\
&\pi \models \phi_1 U \phi_2 && \iff \exists i \geq 0 \bullet \pi_{[i\ldots]} \models \phi_2 \wedge \forall j \bullet 0 \leq j < i \\
& && \qquad \bullet\ \pi_{[j\ldots]} \models \phi_1
\end{aligned}$$

where $\pi_{[i\ldots]}$ denotes the suffix of π starting from state i. An FTS fts satisfies an fLTL property $[\chi]\phi$, denoted by $fts \models [\chi]\phi$, iff

$$\begin{aligned}
&\forall p \in [\![d]\!]_{\mathrm{FD}} \cap [\![\chi]\!] \bullet fts_{\,|p} \models \phi \\
&\quad \iff \forall \pi \in [\![fts_{\,|p}]\!] \bullet \pi \models \phi.
\end{aligned}$$

That is, an FTS satisfies an fLTL property if each product included in the quantification and the FD projects to a TS that satisfies the LTL property.

3.3 The model checking problem in SPLs

The SPL model checking problem consists in determining if the behaviours of all the products in the SPL satisfy a certain property. In case some products do not satisfy the property, there are two possible answers. One is to prove non-satisfaction by a counterexample consisting of a product and a violating execution. The other is to compute all violating products and provide a counterexample for each. This leads to two SPL model checking problems: Mc and $ExtMc$, respectively.

Definition 6 Given an fLTL formula ϕ and an FTS fts, $Mc(fts, \phi)$ returns *true* iff $fts \models \phi$. If $fts \not\models \phi$, it returns *false*, a counterexample e, and a non-empty set of products $px \subseteq [\![d]\!]_{\mathrm{FD}}$ such that $\forall p \in px \bullet fts_{\,|p} \not\models \phi$ with e as counterexample.

Definition 7 Given an fLTL formula ϕ and an FTS fts, $ExtMc(fts, \phi)$ returns *true* iff $fts \models \phi$. If $fts \not\models \phi$, it returns *false* and a set c of couples (e, px) where px is a non-empty set of products such that $\forall p \in px \bullet fts_{\,|p} \not\models \phi$ with e as a counterexample. Furthermore, it holds that

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

The Mc SPL model checking problem is a straight generalisation of the model checking problem for single systems. However, as it only provides a counterexample for one of the products, such a model check is only of limited use. Contrary to $ExtMc$, it does not reveal much about the features required for the violation to occur.

In practice, the set of products is expected to be given as a feature expression. For example, when a check returns $f \wedge \neg g$, the property violation can be attributed to features f and g; moreover, this result might mean that f depends on g, and that this dependency is not documented in the FD. When the set of products is returned explicitly, by listing all violating products, the list has to be analysed to obtain information about the responsible features. Furthermore, returning a list of products does not scale with the number of features

A straightforward but rather naïve algorithm to solve these model checking problems would be to check all valid products individually. That is, list the valid products, compute the projection of each, and model check each projected TS using a standard algorithm. In [13,14], we give alternate algorithms for solving these problems, which are more efficient. These algorithms explore the FTS rather than the individual TSs. The algorithms compute for each state the set of products in which it is reachable. Sets of products are kept in a symbolic data structure, resulting in a semi-symbolic model checking algorithm. SNIP implements an on-the-fly version of this algorithm.

Input: $fts = (S, Act, trans, I, AP, L, d, \gamma)$.
Output: The full reachability relation of fts.

```
1  R ← {(s0, P(P(N))) | s0 ∈ I};
2  Stack ← [];
3  while I ≠ ∅ do
4      Take s0 from I;
5      I ← I \ {s0};
6      push((s0, P(P(N))), Stack);
7      while Stack ≠ [] do
8          (s, px) ← top(Stack);
9          new ← { (s', px' \ R(s')) ∈ Post(s, px)
                 | px' ⊈ R(s') ∧ (px' \ R(s')) ∩ [[d]]_FD ≠ ∅ };
10         if new = ∅ then
11             pop(Stack)
12         else
13             Take (s', px') ∈ new;
14             R(s') ← R(s') ∪ px';
15             push((s', px'), Stack)
16         end
17     end
18 end
19 return R
```

Procedure `Reachables(fts)`

Procedure `Reachables` implements an algorithm that computes the reachability relation R. It generalises the standard Depth-First Search (DFS) algorithm used for TSs, by marking states with sets of products, rather than with

Boolean *visited* flags. In addition, our extension uses a generalised $Post$ operator that computes, for a state s reachable by products px, the successor of s:

$$Post(s, px) \triangleq$$
$$\{(s', px') \mid s \xrightarrow{\alpha} s' \in trans \ \wedge px' = px \cap [\![\gamma(s \xrightarrow{\alpha} s')]\!]\}.$$

In contrast to the DFS algorithm for TSs, where no state is visited twice, our algorithm can visit states multiple times. This is due to the fact that reachability is defined with respect to a set of products. When $R(s) = px$ and the DFS arrives at s for the second time with $px' \not\subseteq px$, then s, although already visited, has to be *re-explored*. This is because transitions that were disallowed for px might be allowed in px'.

The algorithm maintains R and a stack of states, the *execution stack*. Line 1 initialises R in a way such that the initial states are reachable in all feature combinations. A DFS is then started for each of them (line 6). At each iteration, the DFS calculates the set new of unvisited successors of the current state (line 9). It uses the $Post$ operator and it filters out states and products that are already in R. It also makes sure that at least one valid product is among the remaining products. If all successors were visited, the procedure backtracks (line 11). Otherwise, it proceeds with one of the successor states, which is added to R (line 15).

As shown in [13], this procedure can be extended to verify an LTL property using the technique of automata-based model checking given in [38]. Given an LTL property ϕ, it consists in constructing a Büchi automaton, $a_{\neg\phi}$, that accepts all the executions that violate ϕ. Büchi automata accept infinite behaviours. The synchronous product of this automaton and the FTS yields the FTS that is explored. Procedure `Reachables` can then be used to find the accepting states and subsequently determine the violating products.

4 Specifying SPLs in SNIP

FTSs and FDs are semantic models which are abstracted by high-level modelling languages. SNIP uses two modelling languages for SPL specification, fPromela, a high-level language for FTSs based on Promela and TVL [11], a textual FD language. In this section, we introduce the syntax of fPromela and TVL. We then discuss the semantics of fPromela in terms of FTSs and study its expressiveness.

4.1 Syntax

The syntax of fPromela is almost the same as the one of the Promela [23] specification language used in the well-known and widely used model checker SPIN. This should result in a very gentle learning curve for people that are already familiar with Promela. Consider the following system consisting of a sender and a receiver.

Listing 1 A Promela model with a sender and a receiver.

```
chan buffer = [3] of { int };                    1
                                                 2
active proctype sender() {                       3
  int p;                                         4
  do :: true;                                    5
        if :: p = 0;                             6
           :: p = 1;                             7
        fi;                                      8
        buffer!p;                                9
     od;                                         10
}                                                11
active proctype receiver() {                     12
  do :: true;                                    13
        buffer?_;                                14
     od;                                         15
}                                                16
```

The key elements in Promela and fPromela are processes that describe behaviours of single units in the design. Such processes are specified with the `proctype` keyword. A process has to be started by another process or declared `active`, which means that it is active in the initial state of the system. If several processes are active, their executions are interleaved. The sender and the receiver are both active processes. Processes can communicate through shared variables, or more explicitly through *channels*. The global variable `buffer` is a channel of integers with a capacity of three. This means that it can hold three messages. A channel will refuse new messages to be sent once it is full; it will also refuse being read from once it is empty.

A process can have local variables which cannot be accessed by other processes, e.g. variable `p` defined at line 4 of the above code. The behaviour of a process is specified in a procedural style. The `do` statement is used to declare a loop. A `do` loop can have several loop bodies (called *options*), each introduced with a double colon, `::`. The first statement of an option is the condition under which it can be executed. The loop at line 5 has one option with the condition `true`, which means that it can always be executed, effectively making the loop infinite.[2] `if` statements work similarly. The `if` statement at line 6 has two options. Their first and only statements are assignments, which can always be executed. When there are several options that can be executed like this, the choice is non-deterministic. The `if` statement at lines 6–8 is thus used to assign a value to `p` non-deterministically.[3]

Unlike in a programming language, execution of a statement in Promela halts (or *blocks*) until the statement can be executed. At line 9, the content of the variable `p` is

[2] A loop can only be left with the `break` statement, even if it is not finite. When the conditions of all its options are $false$, execution will be blocked until one of them becomes $true$.

[3] When all conditions of an `if` statement are $false$, execution will be blocked until one of them becomes $true$. To avoid this, the `else` expression can be used in one of the conditions.

written into the channel `buffer`. If the channel is full, execution will be blocked here until there is space in the channel. In essence, the sender process non-deterministically writes zeros and ones into a buffer. The receiver process indefinitely reads from the buffer (line 15). When reading from a channel, the underscore means that a message is discarded. If a variable is used, e.g., `buffer?var`, the message is written into the variable. In both cases, the message is removed from the channel, which frees up space.

Promela has a rather extensive syntax that is known to be richer than those used in other model checkers. This above introduction only covered the most important constructs of the language. However, it is worth mentioning that almost all constructs that exist in Promela are available in fPromela and SNIP, too. A full list of unsupported constructs is distributed with SNIP and listed on the website [9].

Let us now switch from Promela to fPromela. fPromela extends Promela with a new type, *feature variables*. Feature variables can be used to *guard* statements with *feature expressions*. The following example illustrates this.

Listing 2 A simple fPromela model.

```
1  // Declare features
2  typedef features {
3    bool Foo;
4    bool Bar
5  };
6  features f;
7
8  active proctype toto() {
9    int i = 0;
10   // Guarded increment statement
11   gd :: f.Foo || f.Bar;
12         i++;
13      :: else;
14         skip;
15   dg;
16   // Test assertion
17   assert(i == 1);
18 }
```

The features used in a model have to be declared as fields of the special type `features`, which is done at lines 2–6. The reason for this is twofold: it serves as an interface that identifies the features used in the model and it ensures compatibility with Promela. The features can then be referenced by declaring any variable with this type (`f` in the example).

The example system consists of one process, specified at lines 8–18. Variability in fPromela is expressed by *guarding* statements. Guard blocks use the `gd` keyword. As an example, the `i++` statement at line 12 is guarded with the expression `f.Foo || f.Bar` (line 11). This means that the `i++` statement is only part of products containing features *Foo* or *Bar*. The other products (line 13) do nothing (line 14). A `gd` statement works like an `if` statement, except that only feature variables or the `else` keyword can be used in the first statement (the condition) of its options. In fact, this is the only place where feature variables may be used. They cannot be accessed anywhere else, be written to or printed. In the language of the C preprocessor, the above guard would have been written as follows:

```
#if defined(FOO) || defined(BAR)   1
  i++;                             2
#endif                             3
```

where `FOO` and `BAR` are directives that are set at compile time if the corresponding features are to be included. Variability in fPromela is thus expressed in a way that is very similar to how it would be expressed in a programming language such as C. Like in C, any fPromela statement can be guarded and guards can be nested. However, unlike `#ifdefs`, the `gd` statements in fPromela are part of the language and its grammar. This way, we avoid the problems that exist when parsing C code with `#ifdefs` [19,27], and any product is guaranteed to be syntactically correct. Note that the `else` in fPromela has to be specified (it is not required by the C preprocessor). Otherwise, execution of all other products will be blocked at this point (which is consistent with the `if` in Promela).

It is worth mentioning that directives of the C preprocessor can also be used in fPromela. However, they cannot be used to specify variability, but only to simplify the model, define constants, decompose it into several files, and so on. This is very helpful for specifying properties, and is required to make SNIP compatible with SPIN. Further note that `gd` and `dg` are just aliases for `if` and `fi`. SNIP will distinguish guards from normal `if`s on its own. This way, a syntactically valid and well-typed fPromela file is also a syntactically valid and well-typed Promela file, provided that all `gd`s are replaced by `if`s.

Going back to the example from Listing 2, we see that a guarded statement is part of the model of a product if its guard evaluates to *true* in the product. In the example, this means that `i` is only incremented in products containing features `f.Foo` or `f.Bar`. At line 17, the property that `i` equals one is specified using an assertion. Alternatively, properties can be specified using LTL, fLTL or directly as automata (that is, using *never claims* as in SPIN). Details regarding how properties are specified will be provided in the next section.

4.2 TVL

TVL [11] is a textual FD language. SNIP requires that all features declared in an fPromela file exist in an accompanying TVL file. For example, a TVL feature model for the previous example would be the following.

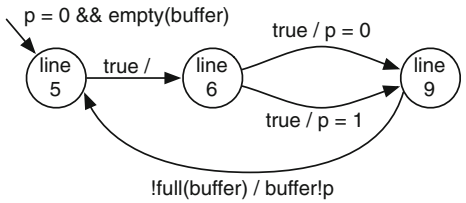


Fig. 1 Program graph of the sender process from Listing 1

Listing 3 FD specified in TVL

```
root Example group allOf {
  opt Foo,
  opt Bar
}
```

An FD is a directed acyclic graph; generally a tree. The `root` keyword denotes the root of the tree, whereas `group` declares branches. In this case, the root feature is called *Example*, with two optional (specified by `opt`) child features *Foo* and *Bar*. The keyword combination `group allOf` means that if the parent feature is part of a product, all of its non-optional children have to be as well. Other decomposition keywords are `someOf` (one or more of the non-optional children have to be part of the product) and `oneOf` (exactly one of the non-optional children have to be part of the product). The FD of Listing 3 represents a set of four products: {*Example*}, {*Example*, *Foo*}, {*Example*, *Bar*}, {*Example*, *Foo*, *Bar*}.

FDs can be encoded as Boolean functions over the features (i.e., *feature expressions*) [5] and analysed with SAT solvers or binary decision diagrams [7].

4.3 Semantics

The syntax of Promela (and hence fPromela) is rather vast, so that it would be tedious to define its full semantics here. We thus omit the less relevant details of the semantics. The interested reader is referred to [23], which contains a precise account of Promela's semantics (in fact, [23] is the only reference used for the implementation of SNIP).

4.3.1 Program graphs

To clarify what a Promela or fPromela model represents, we define the abstract syntax of both languages. Each proctype of a model defines a *program graph*.[4] As an example, the program graph corresponding to the sender process of Listing 1 is shown in Fig. 1.

A program graph is defined over a set of typed variables. The vertices of this graph are the control locations (i.e., the program counter, represented by the line number in Fig. 1) and its transition relation defines the control flow. Each transition has a condition under which it can be executed, and an effect, i.e., a function that defines its effect on the set of variables. In Fig. 1, the transitions are annotated with *condition/effect*. In Promela, the control statements such as `do`, `if` or the semicolon define the control flow. The only statements that end up on transitions are expressions, assignments (including channel reads and writes), assertions and print statements (not shown in our examples, but supported by SNIP). For the purpose of this discussion, we only consider expressions and assignments.

Definition 8 Let $types$ be the set of types in Promela, $V = \{v_1, \ldots, v_k\}$ a set of variables, and $\tau : V \to types$ their type function: $expr(V)$ denotes all Promela expressions over V, and $asgn(V)$ all assignments. Assuming that the variables are ordered, $v \in \tau(v_1) \times \cdots \times \tau(v_k)$ denotes a valuation of the variables, let $val(V)$ be the set of all valuations. For $e \in expr(V)$, we write that $v \models e$ if the expression evaluates to $true$ for the values v. For $a \in asgn(V)$, $apply(a, v) \in val(V)$ denotes the valuation obtained after applying the assignment a to v.

We define a program graph as a graph in which each transition is labelled with an expression (the condition) and an assignment (its effect). If a statement in a model has no effect on the variables, its assignment is the identity function. If a statement can be executed at all times, its condition is simply $true$. In addition, a program graph has an expression characterising the variable values in the initial state.

Definition 9 A program graph over (V, τ) is a tuple $(S, trans, I, init)$, where S is a set of states and $I \subseteq S$ a set of initial states, $trans \subseteq S \times expr(V) \times asgn(V) \times S$ is the transition relation, and $init \in expr(V)$ is an expression characterising the variable values in the initial state.

The semantics of a program graph G, noted $[\![G]\!]$, is a TS $(S', Act', trans', I', expr(V), L')$ where

- $S' = S \times val(V)$, that is, each state denotes a control location and a variable valuation of the program graph;
- $I' = \{(i, v) \mid i \in I, v \in val(V) \bullet v \models init\}$, the initial states are the initial control locations and valuations satisfying $init$;
- $Act' = \{\epsilon\}$, actions are not required here, so each transition is labelled with a dummy action ϵ;
- $L'\big((s, val)\big) = \{e \in expr(V) \mid val \models e\}$, each state is labelled with the expressions satisfied by its variable valuation;
- Transitions can only be executed when the expression evaluates to $true$ for the variable valuation in the start state; they change the variable valuation in the end state

[4] Holzmann uses the term 'finite state automaton' [23]. We use 'program graph' [4].

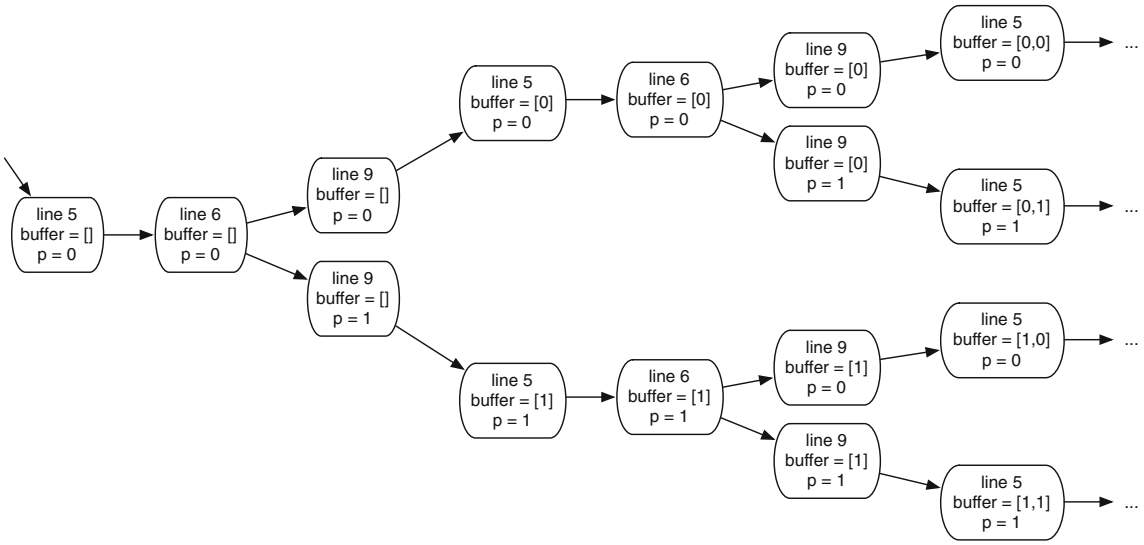


Fig. 2 TS corresponding to the program graph of Fig. 1

according to the assignment: for $e \in expr(V)$, $a \in asgn(V)$ and $v \in val(V)$,

$$\frac{(s, e, a, s') \in trans \wedge val \models e}{((s, val), \epsilon, (s', apply(a, val))) \in trans'}.$$

The program graph corresponding to a Promela file is obtained by transforming control statements into vertices. In a nutshell, this is done as follows. Single expressions terminated by a semicolon correspond to states with single outgoing transitions. An example is the transition from *line 5* to *line 6* in Fig. 1. An `if` statement becomes a state with one outgoing transition per option. This transition is labelled with the first expression of the option. The last state of all options leads back to a common state. In the example of the sender, there is just one transition per option, which is why both transitions leaving *line 6* lead to the same state in Fig. 1. A `do` statement is similar to an if statement, except that the last transition of all options leads back to the beginning of the `do` statement. Assignment statements are always executable, their condition is thus $true$. Channel writes are only executable if the channel is not full. In Promela, variables are implicitly initialised at zero and channels are empty; hence the expression on the initial transition in Fig. 1.

A fragment of the TS corresponding to the program graph of the sender is shown in Fig. 2. The TS has no infinite behaviours. Every behaviour stops in a state where the control location is in *line 9* and the buffer contains three elements. The buffer is thus full and execution blocks at *line 9*. To obtain a system with no finite behaviours, the program graph of the sender and the one of the receiver have to be put in parallel. The parallel composition of two program graphs is obtained by interleaving their executions. One exception are *rendezvous channels*, which are channels of capacity zero. Reads and writes of rendez-vous channels have to occur together, at the same time, which means that these transitions are synchronised. The parallel composition of two program graphs results in a program graph over the union of their variables. We do not go into the details of this definition. It is very similar to the classical definition of parallel composition of TSs, except that shared variables are taken into account. The interested reader is referred to [4, Sects. 2.2.2 and 2.2.4].

As expected, the semantics of a Promela model is a TS. The actual semantics is much more intricate, but Definition 9 captures the gist of it.

4.3.2 Featured program graphs

An fPromela model describes a *featured program graph*. A featured program graph is a program graph in which transitions are annotated with feature expressions.

Definition 10 A featured program graph over variables (V, τ), and an FD d with features N, is a tuple $(S, trans, I, init, \gamma)$, where S, I, $trans$ and $init$ are defined as in Definition 9 and $\gamma : trans \to \mathbb{B}(\{f_1, \ldots, f_n\})$ annotates transitions with a feature expression.

The semantics of a featured program graph is an FTS $(S', \{\epsilon\}, trans', I', expr(V), L', d, \gamma')$ where S', I' and $trans'$ are defined as in Definition 9. Note that there is a clear correspondence between transitions of the program graph and those of the TS. In the FTS, γ' is such that for any transition $t' \in trans'$, let $t \in trans$ be the corresponding transition of the program graph, then $\gamma'(t') = true$ if $\gamma(t)$ is not defined (i.e., for an unguarded transition) and $\gamma'(t') = \gamma(t)$ otherwise.

The featured program graph of an fPromela file is obtained in a way similar to obtaining the program graph of a normal

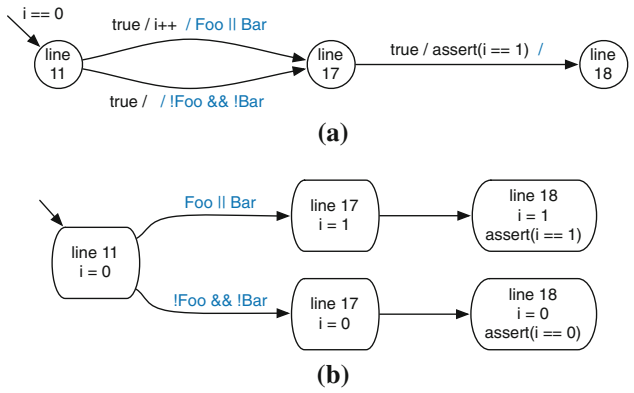


Fig. 3 The fPromela example from Listing 2. **a** The featured program graph of the model. **b** The corresponding FTS

Promela file. The only difference is the treatment of `gd` statements. The featured program graph of a Promela file would be equivalent to its program graph. Remember, the first statement of an option of a `gd` statement is the feature expression. The other statements are the guarded behaviour. In terms of the featured program graph, this means that a `gd` statement is treated like an `if` statement, except that the first statement acts as the feature expression of the *second* statement. If the first statement of an option block is `else`, the feature expression is the negation of the conjunction of the feature expressions of the other options in the `gd` statement.

To illustrate this, the featured program graph corresponding to the fPromela example from Listing 2 is shown in Fig. 3a. The feature expression labels were added in colour behind the existing labels. All unguarded statements, like the `assert` at line 17, are not labelled. The corresponding FTS is shown in Fig. 3b.

The semantics of an fPromela model is an FTS. As for Promela, the actual semantics is much more intricate than Definition 10. In fact, it follows the semantics of Promela (given in [23]) exactly, just adding feature expressions from the featured program graph to the transitions. The parallel composition of two featured program graphs is defined in the same way as for program graphs. Each transition of the resulting featured program graph corresponds to a transition of one of the featured program graphs, whose feature expression it inherits. An exception are rendez-vous transitions, whose feature expression are the conjunction of those of the transitions being executed in parallel.

Just like FTSs and TSs, featured program graphs and program graphs are related by a projection operation. The following definition is analogous to Definition 3.

Definition 11 Given a featured program graph $fG = (S, trans, I, init, \gamma)$ with FD d, its projection to a product $p \in [\![d]\!]_{\text{FD}}$, noted $fG_{|p}$, is the program graph $(S, trans', I, init)$ where

$trans' \triangleq \{t \in trans \mid t \notin dom(\gamma) \vee p \models \gamma(t)\}$.

This definition covers the case in which $\gamma(t)$ is not defined (e.g., for an unguarded transition). If a transition is not in the domain of γ, $dom(\gamma)$, the transition is included.

Syntactically, the projection operation can be accomplished as follows.

Algorithm 1 Given an fPromela model with FD d, its projection to a product $p \in [\![d]\!]_{\text{FD}}$ can be obtained as follows:

(1) remove all feature variable declarations;
(2) replace the feature expressions of all `gd` statements by the value they take for p;
(3) remove the feature expressions that evaluate to $true$;
(4) replace all `gd` statements by `if` statements.

The obtained model is a syntactically valid Promela model. Since feature variables are guaranteed to only appear in feature expressions, removing them will not lead to bad references.

Theorem 1 *Given an fPromela model with FD d and featured program graph fG, the reduction to a product $p \in [\![d]\!]_{\text{FD}}$ computed according to Algorithm 1 yields a Promela model whose program graph is semantically equivalent to $fG_{|p}$.*

Proof It should be clear that without steps (2) and (3) of Algorithm 1, the resulting program graph would have at least the transitions and states of $fG_{|p}$. The effect of step (2) is that all feature expressions that evaluate to $false$ become transitions which can never be executed. This is equivalent to removing them. Furthermore, the transitions in fG whose feature expression evaluated to $true$ (that are thus in $fG_{|p}$) are now all prefixed with a single transition with the expression $true$ (the evaluated feature expression). The effect of step (3) is to remove these transitions. The resulting program graph is thus indeed semantically equivalent to $fG_{|p}$. □

An immediate consequence of the above result is that an fPromela model without any guarded statements can be interpreted either as an fPromela or as a Promela model. The first interpretation yields an FTS and the second yields a TS, both of which have exactly the same behaviours. For the fPromela example from Listing 2, the projection to the product $\{Example, Foo\}$ is the following:

```
active proctype toto() {                    1
  int i = 0;                                2
  i++;                                      3
  assert(i == 1);                           4
}                                           5
```

The relation between fPromela and Promela is thus very similar to that between FTSs and TSs, as is the relation between fPromela and FTSs and Promela and TSs. This is

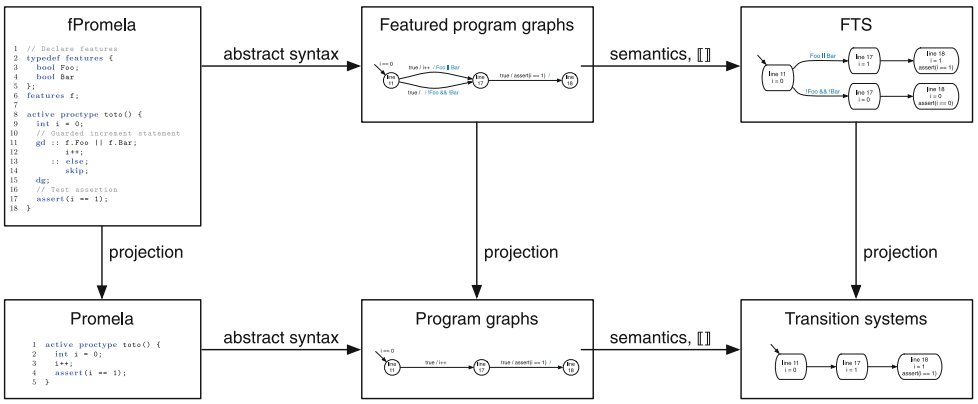


Fig. 4 Relation of fPromela, Promela, FTSs and TSs

best illustrated by the diagram in Fig. 4. The diagram is commutative, as Theorem 1 and the following theorem establish.

Theorem 2 *For any fPromela model with with FD d and featured program graph fG, for any product $p \in [\![d]\!]_{\text{FD}}$,*

$$[\![fG_{|p}]\!] \equiv [\![fG]\!]_{|p}$$

where $\equiv$ means that both TSs are semantically equivalent.

The semantics of a program graph is defined in the same way as the semantics of a featured program graph. Furthermore, feature expressions are treated in the same way by FTS projection (Definition 3) and by the projection of a featured program graph (Definition 11). In consequence, the order in which these operations are applied does not matter. Hence, $[\![fG_{|p}]\!]$ is syntactically equivalent to $[\![fG]\!]_{|p}$.

This concludes our discussion of the semantics of fPromela. Before we proceed to study its expressiveness, we would like to point out that there is also a much easier method to implement syntactic projection.

Algorithm 2 Given an fPromela model with FD d, its projection to a product $p \in [\![d]\!]_{\text{FD}}$ can be obtained as follows:

(1) initialise all feature variables according to p;
(2) replace all `gd` statements by `if` statements.

The advantage of this method is that it does not require the whole fPromela file to be parsed. As `gd` can be replaced by `if` in an fPromela file anyway, all that needs to be done is to initialise the feature variables. For the example from Listing 2, this would yield the following.

```
typedef product {                          1
  bool Foo = 1;     // initialise           2
  bool Bar = 0      // initialise           3
};                                         4
product f;                                 5
                                           6
active proctype toto() {                   7
  int i = 0;                               8
  // gd becomes if                         9
  if :: f.Foo || f.Bar;                    10
        i++;                               11
     :: else;                              12
        skip;                              13
  fi;                                      14
                                           15
  assert(i == 1);                          16
}                                          17
```

The principal difference with Algorithm 1 is that the feature expressions at line 10 and line 12 remain as transitions. This changes nothing for those that evaluate to $false$ since they can never be taken. Those that evaluate to $true$, however, are additional transitions that have to be executed before the actual guarded statement is executed. This leads to two discrepancies. First, these transitions will create *stutter steps* in the underlying TS (i.e., transitions that do not modify the atomic propositions).

Definition 12 A *stutter step* is a transition $s_1 \xrightarrow{\alpha} s_2$ such that $L(s_1) = L(s_2)$. Two executions are *stutter equivalent* if they only differ in their stutter steps, and two TSs ts_1 and ts_2 are *stutter trace equivalent* if for any execution in $[\![ts_1]\!]_{TS}$, there exists a stutter equivalent execution in $[\![ts_2]\!]_{TS}$, and vice-versa. By extension, two program graphs are stutter trace equivalent if their TSs are stutter trace equivalent.

In addition to creating stutter transitions in the underlying TS, these transitions might also introduce new non-determinism. Consider the following case:

```
gd :: f.Foo;                               1
      chan!1;                              2
   :: true;                                3
      skip;                                4
dg;                                        5
```

The channel write statement is only part of products with feature Foo, whereas the `skip` is part of all products. A discrepancy arises in products with feature Foo. First, consider

projection according to Algorithm 1: when the execution gets to the above statements and the channel is full, the system will always take the `skip` transition. If projected according to Algorithm 2, the channel write will be prefixed by a `true` transition. Now, when the execution gets to the guarded statement, the system has the non-deterministic choice of taking the `skip` or the `true` transition. If the `true` transition is taken, the execution will be blocked at the channel write statement. If the channel remains full indefinitely, this leads to a deadlock which does not exist in the other projection. The problem is that the `true` transition introduced non-determinism which did not exist in the actual system. A necessary condition for this problem to occur is that the guarded statement is not *exclusive*, i.e., some of the feature sets defined by its feature expressions overlap. Otherwise, the `true` transition would be the only one, and thus cannot introduce new non-determinism. This is formalised by the following theorem.

Theorem 3 *Given an fPromela model with FD d and featured program graph fG, the reduction to a product $p \in [\![d]\!]_{\text{FD}}$ computed according to Algorithm 2 yields a Promela model whose program graph G is* stutter trace equivalent *to $fG_{|p}$ if all `gd` statements are* exclusive. *Exclusive means that the feature sets defined by the feature expressions of a `gd` statement are disjoint.*

Proof In the resulting Promela model, the feature variables are normal Boolean variables. By definition of fPromela, they are never written to, which means that all feature expressions will always evaluate to the same value. The resulting program graph G thus corresponds to that of Algorithm 1 in which step (3) was not executed. As stated in the proof of Theorem 1, the transitions in fG whose feature expression evaluated to $true$ are now prefixed with a transition (the former feature expression) whose expression always evaluates to $true$. These $true$ transitions lead to a stutter transition in $[\![G]\!]$, since they do not change the variables. However, $[\![G]\!]$ is only stutter equivalent to $[\![fG_{|p}]\!]$ if the $true$ transitions do not introduce new non-determinism. If the feature sets of a `gd` statement are disjoint, it will have at most one $true$ transition in any product. In this case, the $true$ transitions do not introduce new non-determinism. □

In practice, the feature expressions of a `gd` statement are almost always disjoint. Furthermore, stutter equivalence preserves all LTL properties that do not use the next operator ($\bigcirc$), which is almost never used. Because of this, and because of its ease of implementation, we use Algorithm 2 to implement projection (the input to SPIN) in our benchmarks.

The above theorem has a corollary which yields an alternative, much more intuitive, semantics for fPromela.

Corollary 1 *Each fPromela model is semantically equivalent to the non-deterministic choice between 2^n Promela models (where n is the number of feature variables) obtained by varying the initial values of the features. (Provided that the feature expressions of all `gd` statements are disjoint.)*

4.4 Expressiveness

By Definition 10, any fPromela model can be translated into an equivalent FTS. The fPromela modelling language is thus a subset of the FTS language. It is not difficult to show that the converse holds as well, i.e., that both languages are expressively equivalent.

Theorem 4 *Any FTS can be translated into fPromela.*

Proof Let $(S, Act, trans, I, AP, L, d, \gamma)$ be an FTS. An equivalent fPromela model with FD d can be obtained by encoding the transition relation with `goto` statements (which work similar to C). Basically, each state becomes a program location and `goto`s are used to jump from location to location reflecting the transition relation. Each `goto` thus corresponds to one transition and is guarded with the feature expression of the transition. For each state $s \in S$ with its outgoing transitions, this yields:

```
// One label identifying the state:
state_s:
// If the state is an initial state,
   a second label:
init_s:
  // For each target state one option
     with a goto:
  gd :: feature expression;
        goto state_target;
     :: ...
     dg;
```

The initial states are modelled as follows:

```
// One goto per initial state
if :: goto init_state;
   :: ...
   fi;
```

The program graph G of an active `proctype` with this behaviour will have $|S| + 1$ control locations, one for each state, plus the additional initial state. Since there are no variables, $[\![G]\!]$ is syntactically identical to the input FTS, except for the additional initial state. □

5 SNIP

We now focus on the usage of SNIP. First, we present the user interface of the tool that we illustrate with several examples. We then discuss the architecture of SNIP as well as various implementation choices and third-party libraries used

in its design. Finally, we describe how the FTS algorithms of [13,14] have been implemented.

5.1 User interface and illustration

The user interface of SNIP is designed to take into account the various SPL model checking use cases. It also addresses a variety of practical concerns such as simulation, bounded checking, or layout of counterexamples.

Since our development has focused mainly on the model checking algorithms, SNIP is currently a command-line application, with no graphical user interface. However, we do not believe that a model checker needs to have a graphical user interface to be user-friendly. As for most model checkers, SNIP's use consists in launching checks with certain parameters (property, execution bound). The command line is a very efficient and convenient interface for this kind of task. It remembers past commands and keeps a trace of inputs and outputs.

To make it easy to get started with SNIP, the list of its parameters is shown when launching SNIP without any parameter (or with bad parameters).

5.1.1 Introductory example and assertion checking

As inputs, SNIP requires an fPromela file, a TVL file and a property. For our first illustration, we use the example from the previous section, where the fPromela file is given in Listing 2 and the TVL file in Listing 3. In this case, the property is the assertion at line 17 of the model. To check it, SNIP would be executed as follows.

```
$ ./snip -check -fm features.tvl model.pml
No never claim, checking only asserts and deadlocks..
Assertion at line 17 violated [explored 5 states,
re-explored 0].
 - Products by which it is violated (as feature
   expression):
   (!Foo & !Bar)

 - Stack trace:
   features                          = /
   pid 00, toto                      @ NL11
   toto.i                            = 0
    --
   features                          = (!Foo & !Bar)
   pid 00, toto                      @ NL14
    --
   features                          = (!Foo & !Bar)
   pid 00, toto                      @ NL17
    --
    -- Final state repeated in full:
   features                          = (!Foo & !Bar)
   pid 00, toto                      @ NL17
   toto.i                            = 0
    --
```

The `-check` parameter activates SNIP's model checker. If it is set, SNIP automatically checks all assertions and looks for deadlocks. The `-fm` parameter specifies the feature model. This parameter can be omitted if the TVL file has the same name as the fPromela file. That is, if the TVL file were named `model.tvl`, the preceding command-line can be shortened to:

```
$ ./snip -check model.pml
```

The output provided by SNIP consists of two parts. First, SNIP reports the products for which the property is violated in the form of a feature expression (line 5). Second, SNIP gives a counterexample, that is, an execution of the fPromela model which proves the property violation (line 7 and following). It is presented as a sequence of states separated by double dashes. For each state, SNIP prints the products that can reach the state as a feature expression ('`/`' means all products), the position inside each process (`pid 00, toto @ NL11` means the process with id 00, of type `toto` is at line 11), and the values of the variables. At line 3, SNIP also reports two statistics: the number of states that were explored and re-explored. The explored states are the states that were visited and stored in memory; the re-explored states are visited states that had to be explored again.

To make counterexamples shorter and easier to understand, variables are only printed if their value has changed. Furthermore, the last state is repeated in full so that the user can work backwards. There are two options to control the output of counterexamples: `-nt` disables them (very useful if the user is only interested in the satisfying products), and `-st` prints only states in which variable values changed (i.e., states in which processes do nothing are not shown.). Since SNIP's output is text and can be interpreted immediately (no need for an additional tool), it can be piped to other command-line tools such as *cat* or *grep*. This is very useful to filter the relevant variables out of long counterexamples.

For the example, SNIP reports that the assertion is violated by products that satisfy `!Foo & !Bar`. This is as expected, since only those products lack the `i++` statement at line 12. SNIP implements both MC and EXTMC. If only `-check` is specified, SNIP computes MC. This means that SNIP stops as soon as it finds a violation. To compute EXTMC, the parameters `-check` and `-exhaustive` have to be set.

```
$ ./snip -nt -check -exhaustive -fm features.tvl
model.pml
No never claim, checking only asserts and deadlocks..
Assertion at line 17 violated [explored 5 states,
re-explored 0].                                          5
 - Products by which it is violated (as feature
   expression):
   (!Foo & !Bar)

Exhaustive search finished  [explored 5 states,          10
re-explored 0].
 - One problem found covering the following products
   (others are ok):
(!Foo & !Bar)
```

In this case, SNIP will print a violation upon finding it (line 4), and continues searching for violations in the other products. In the example, we disabled printing of

counterexamples using `-nt`, otherwise, SNIP will print a counterexample for each violation. When the search terminates, SNIP prints a summary with all the products found to violate (line 14). In this case, those are the same as before. However, we now have the certitude that all products satisfying `Foo | Bar` are free from violations.

5.1.2 Sender/receiver example and deadlock checking

We now consider a more complex example that uses both parallel composition and infinite behaviours. For doing so, we modify the sender/receiver example given in Listing 1 by making each of the two processes optional. The FD in this case would be the following.

```
root Main group someOf {
  Send,
  Receive
}
```

The Promela model is transformed into the following fPromela model.

```
typedef features {
  bool Send;
  bool Receive
};
features f;

chan buffer = [3] of { int };

proctype sender() {
  int p;
  do :: true;
        if :: p = 0;
           :: p = 1;
        fi;
        buffer!p;
     od;
}

proctype receiver() {
  do :: true;
        buffer?_;
     od;
}

active proctype boot() {
  gd :: f.Send;
        run sender();
     :: else;
        skip;
     dg;
  gd :: f.Receive;
        run receiver();
     :: else;
        skip;
     dg;
}
```

Instead of declaring the sender and the receiver processes `active`, they are now started explicitly by the `boot` process, using Promela's `run` statement. Each `run` statement is guarded by the respective feature. This way, only products with the *Send* feature have a sender process, and likewise for the *Receive* feature. Checking this model yields the following.

```
$ ./snip -check -exhaustive -nt sendrcv.pml
No never claim, checking only asserts and deadlocks..
Found deadlock [explored 139 states, re-explored 0].
 - Products by which it is violated (as feature
   expression):                                          5
   (Send & !Receive)

Found deadlock [explored 202 states, re-explored 0].
 - Products by which it is violated (as feature
   expression):                                         10
   (!Send & Receive)

Exhaustive search finished  [explored 202 states,
re-explored 0].
 - 2 problems were found covering the following        15
   products (others are ok):
(!Send & Receive) | (Send & !Receive)
```

SNIP finds two deadlocks, as expected. The counterexamples (disabled here for brevity) identify the deadlocked states. In the first case, the sender is started without a receiver. It will thus send messages until the buffer is full, at which point it waits indefinitely at line 15; a deadlock state. In the second case, the receiver is started without a sender. It deadlocks immediately at line 21 because the buffer will always remain empty.

It might seem that this is inconsistent with the previous example. It too has only finite behaviours, and yet SNIP did not report a deadlock. This is because its finite behaviours all end with a terminal state of the program graph (in this case, the end of the process specification). In Promela and fPromela, a state with no outgoing transitions is not a deadlock state if all processes are in terminal states. In the deadlock states of the sender/receiver example, the `boot` process is in a terminal state, whereas the `sender` (or `receiver`) is not.

In FTSs, deadlocks can also stem from erroneous feature expressions. Consider the following example in which *A* is a single optional feature.

```
typedef features {
  bool A
};
features f;

active proctype foo() {
  int i = 0;
  gd :: f.A;
        i++;
  dg;
  i++;
}
```

The guard at line 8 only considers products with feature *A*. For all other products, there will be no transition in this state. Those products are deadlocked, as the `foo` process is blocked in a non-terminal state. In SNIP, such special deadlocks states are called *trivially invalid end states*; 'trivially', because they can be very easily avoided by making sure each `gd` statement has an `else` option. By default, SNIP will not check for trivially invalid end states. In contrast to simple deadlock checking, it requires a small computation each time, which might be costly. If SNIP is run normally, this yields.

```
$ ./snip -check -nt deadlock.pml
No never claim, checking only asserts and deadlocks..
No assertion violations or deadlocks found [explored
2 states, re-explored 0].
```

To activate checking of trivially invalid end states, the `-fdlc` parameter has to be set.

```
$ ./snip -check -exhaustive -fdlc -nt deadlock.pml
No never claim, checking only asserts and deadlocks..
Found trivially invalid end state; the following set
of products can reach the state, but has no outgoing
transition. [explored 1 states, re-explored 0].      5
 - Products by which it is violated (as feature
   expression):
   (!A)
```

SNIP then correctly identifies the products without feature *A* as violating.

5.1.3 Mine pump example and fLTL model checking

So far, we have shown how assertions and deadlocks are checked. Of course, properties can also be specified using LTL, fLTL or directly as *never claims* (an automata-based encoding of the property).

To illustrate this verification procedure, we use the mine pump system, a specification exemplar for distributed systems originally introduced in [29]. The purpose of the system is to keep a mine shaft clear of water while avoiding the danger of a methane explosion. It consists of a water pump, a sensor measuring the water level and a sensor measuring the concentration of methane in the mine. The system should activate the pump once the water level reaches a preset threshold, but only if the methane is below a critical limit.

The system consists of three high-level features: (i) a command interface *Command*, which can be used to switch the water regulation function on or off; (ii) a methane alarm interface *MethaneSensor*, which can receive alarm messages from the methane sensor in case of critical methane, and (iii) the water regulation subsystem *WaterSensor*. The system is distributed: the controller and sensors are individual subsystems which communicate by message passing. Although the system was not designed as an SPL, these components play the same roles as features in an SPL and can be modelled as such.

We created an fPromela model that follows the CONIC code included in [29] very closely. The model consists of about 200 lines of fPromela. The fPromela model consists of five processes communicating over channels: a controller, a pump, a water sensor, a methane sensor and a user. When activated, the controller should switch on the pump when the water level is high, but only if there is no methane in the mine. The TVL FD is the following.

Listing 4 FD of the mine pump controller product line.

```
root MinePump {                                  1
  group allOf  {                                 2
    opt Command group someOf {                   3
      Start,                                     4
      Stop                                       5
    },                                           6
    opt MethaneSensor group someOf {             7
      MethaneAlarm,                              8
      MethaneQuery                               9
    },                                           10
    WaterSensor group [0..*] {                   11
      Low,                                       12
      Normal,                                    13
      High                                       14
    }                                            15
  }                                              16
}                                                17
```

Most features are self-explanatory. The methane detection is split into two features, corresponding to the two mechanisms used. With the *MethaneAlarm* feature, the controller is notified when there is methane in the mine (it is passive). With the *MethaneQuery*, the controller queries the methane sensor each time before starting the pump (it is active).

The model must satisfy a large set of properties (42). Here, we focus on one such property: *"There is never a situation in which the pump runs indefinitely even though there is methane."*; in LTL this becomes

$$\neg\Diamond\Box(pumpOn \wedge methane)$$

and in the syntax used by SNIP:

```
!<>[] (pumpOn && methane).
```

Checking this property with SNIP yields the following.

The property is specified with the `-ltl` parameter. SNIP finds 16 violations and concludes that all products with `Start & High` violate the property. This is not what we expected, as the property is supposed to be satisfied by the system. Products without *Start* or *High* will never even start the pump, which is why they satisfy the property.

A look at the counterexamples reveals a problem with the property. Basically, the controller has a central loop,

```
$ ./snip -check -exhaustive -nt
         -ltl '!<>[] (pumpOn && methane)'
         minepump.pml
Checking LTL property !<>[] (pumpOn && methane)..
Property violated [explored 481 states, re-explored
0].
 - Products by which it is violated (as feature
   expression):
   (Start & Stop & MethaneQuery & MethaneAlarm & Low
   & High)

[...]

Property violated [explored 12806 states,
re-explored 65409].
 - Products by which it is violated (as feature
 expression):
   (Start & !Stop & !MethaneQuery & !MethaneAlarm &
   !Low & High)

Exhaustive search finished  [explored 17325 states,
re-explored 179937].
 - 16 problems were found covering the following
   products (others are ok):
(Start & High)
```

in which it can receive three types of messages: user commands (start and stop), methane alarm messages, and water level readings. The counterexamples show in every case that the methane sensor sends an alarm message to the controller. However, as the choice of receiving one of the three messages is non-deterministic, the controller might ignore the alarm message indefinitely. In practice, such a behaviour is highly unlikely. It is thus reasonable to assume that the controller will infinitely often accept a message of each type. This assumption can be specified in LTL as follows: `(([]<> readCommand) && ([]<> readAlarm) && ([]<> readLevel))`. If we add it as a premise to the original formula, we obtain the following result.

```
$ ./snip -check -exhaustive -nt
         -ltl '([]<> read..) -> (!<>[] pump..)'
         minepump.pml
Checking LTL property ([]<> read..) ->
(!<>[] pump..)..
Property violated [explored 27428 states,
re-explored 125153].
 - Products by which it is violated (as feature
   expression):
   (Start & Stop & MethaneQuery & !MethaneAlarm &
   Low & High)

[...]

Property violated [explored 30157 states,
re-explored 162316].
 - Products by which it is violated (as feature
   expression):
   (Start & !Stop & !MethaneQuery & !MethaneAlarm &
   !Low & High)

Exhaustive search finished  [explored 34356 states,
re-explored 274456].
 - 8 problems were found covering the following
   products (others are ok):
(Start & !MethaneAlarm & High)
```

This result can be interpreted as saying that the *MethaneAlarm* feature is responsible for making the property true. This corresponds to what we expected, as the *MethaneAlarm* feature alerts the controller of methane, leading it to shut off the pump. However, a product satisfies the property either if it does not satisfy the assumption or if it satisfies the initial property `!<>[] (pumpOn && methane)`. Hence, we have to make sure that there are actually products that satisfy the assumption. This is done by checking its negation.

```
$ ./snip -check -exhaustive -nt
         -ltl '!(([]<> read..)' minepump.pml
Checking LTL property !(([]<> read..)..
Property violated [explored 2169 states, re-explored
0].
 - Products by which it is violated (as feature
   expression):
   (Start & Stop & MethaneQuery & MethaneAlarm & Low
   & High)

[...]

Exhaustive search finished [explored 8091 states,
re-explored 37323]
 - 38 problems were found covering every product.
```

The assumption is thus discharged by all products. Then, we conclude that for all products with the *MethaneAlarm* feature, there exists no path verifying the assumption but violating the property. In simple terms, the feature ensures that the pump will not run indefinitely even though there is methane, as long as the controller does not ignore the alarm messages indefinitely.

Normally, the example property is not expected to hold for products that do not have the *MethaneAlarm* feature. It corresponds to a requirement implemented by the feature. This can be expressed with a quantifier in fLTL:

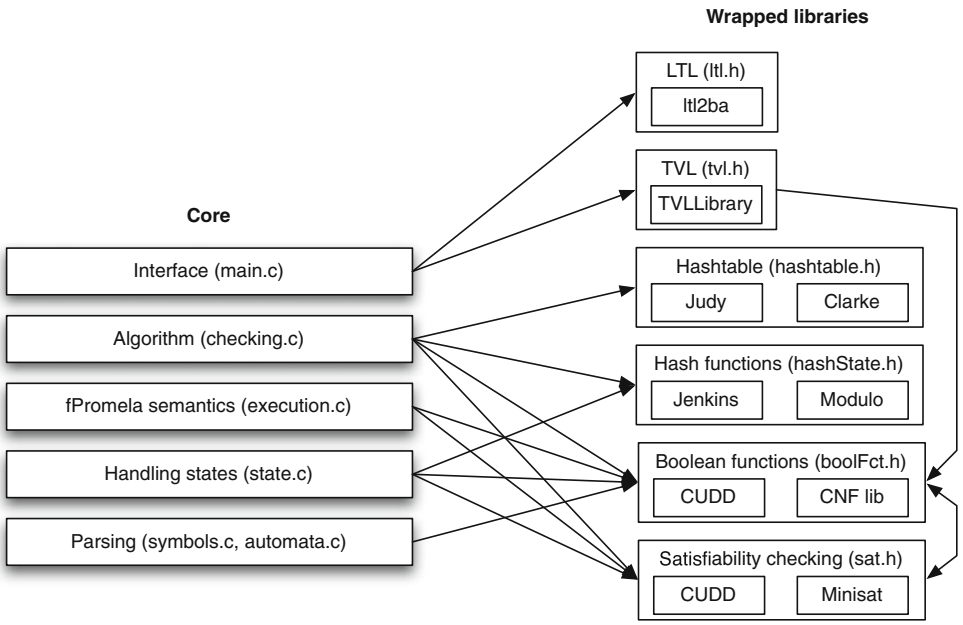
$$[MethaneAlarm]\ \neg\Diamond\Box(pumpOn \wedge methane).$$

In SNIP, the quantifier of an fLTL property is specified in TVL syntax, separately from the LTL property with the `-filter` parameter.

```
$ ./snip -check -exhaustive -nt
         -filter 'MethaneAlarm'
         -ltl '([]<> read..) -> (!<>[] pump..)'
         minepump.pml
Checking LTL property ([]<> read..) ->
(!<>[] pump..)..
Attention! Checks are only done for products
satisfying:
  MethaneAlarm
Property satisfied [explored 27893 states,
re-explored 248254].
```

The property is thus indeed satisfied by all relevant products. SNIP recalls in the output that the property is checked over a subset of the products (line 7).

Fig. 5 Architecture of SNIP



5.2 Architecture and third-party libraries

SNIP is entirely written in the C programming language. An overview of its architecture is shown in Fig. 5. The core of SNIP is divided into layers, so that lower layers are unaware of and have no dependency on upper layers. Each layer has access to a set of wrapped libraries. A wrapper consists of an interface (a header file) against which other code is written, and one or more implementations of the interface depending on the third-party library used. Third-party libraries are thus all wrapped and can easily be replaced. The library to be used for a wrapper is chosen at compile time. Let us first look at the core of SNIP, before we survey the third-party libraries used.

To create the fPromela parser, we use the parser generators Flex and Bison. These tools are highly efficient and the de-facto standard for creating parsers in C. To make sure that a Promela file in SNIP is parsed in the same way as it is in SPIN, we reused the Bison grammar specification from the SPIN source code.[5] As the model is parsed, SNIP fills a symbol table with global variables and process definitions. The body of a process is represented by a featured program graph, which is created at the same time the model is parsed (it is built backwards). We make extensive use of doubly linked lists, one of the primary data structures in SNIP. As the featured program graph is built, all feature expressions are transformed into Boolean function objects (the actual type depends on the library chosen for representing Boolean functions). Furthermore, all references inside expressions are resolved and replaced by pointers to the respective symbols.

The state layer implements functions for the representation and manipulation of system states in memory. SNIP does not use the state compression [22] principle implemented in SPIN. To make state manipulation reasonably efficient nonetheless, all variables are stored in a block of memory (the *payload*), rather than in linked lists. Blocks of memory can be copied and compared efficiently with built-in functions. The payload holds the global variables as well as those of the processes. Since all variables have a fixed size, we only need to keep track of the address in the payload at which the variables of a process start. A state further contains a Boolean function, which characterises the products for which it is reachable. The state layer handles dynamically created processes and channels.

The execution layer implements the semantics of Promela and fPromela. To make sure that the Promela semantics is correctly implemented, we follow the operational Promela reference [23] very closely. We thus use a function `executables`, which determines the transitions that can be executed in a state. It takes the feature expression of the state into account, as well as the feature expressions of the candidate transitions. Other functions that are derived from the Promela specification are `eval`, which evaluates an expression, and `apply`, which executes a transition. The execution layer has also functions for simulation and for managing the execution stack of the DFS.

On top of this hierarchy, we find the model checking layer. It implements the algorithms from [14], both as a nested DFS (for LTL) and as a normal DFS (for asserts and deadlocks). Visited states are stored in a hashtable. The model checking layer will be discussed in more detail in the following section.

[5] This is indeed the only piece of SPIN source code reused in SNIP.

The interface layer pieces all of the other layers together. It is also responsible for the preparatory tasks. More precisely, it interprets the command-line parameters and writes them to global variables. It transforms the LTL property into a Büchi automaton which it appends as a *never claim* to the input file. It then runs the C preprocessor on the input file and launches the parser. The interface layer also guesses the name of the TVL model and transforms it into a Boolean function. A number of temporary files are generated for this, which can be preserved if SNIP is executed with the `-t` parameter.

SNIP uses several third-party libraries. As noted before, the Promela grammar is taken from SPIN. To automate the transformation from LTL to Büchi automata, we use LTL2BA,[6] a very efficient implementation based on the results of [20]. To parse TVL models and transform them into DIMACS, we use the TVL library.[7] DIMCAS is a data exchange format for Boolean functions in CNF. The TVL library is written in JAVA and rather inefficient (even small models take a second to be parsed and transformed). Therefore, SNIP also allows the user to specify the FD in DIMACS directly. For this, SNIP has a command-line switch `-fmdimacs`, which has to be followed by a file in DIMACS format, and a dictionary file. Variable names in a DIMACS file are all integers. The dictionary file lists the feature names of the integers used in the DIMACS file. The TVL library has the ability to export these files.

To store large sets of states with efficient lookup times, we rely on a resizable hash table implementation provided by Clark.[8] Collisions are managed through a self-implemented procedure that uses linked lists.

For the internal representation of Boolean functions, the tool currently offers two alternatives: Binary Decision Diagrams (BDDs) [7] or Conjunctive Normal Forms (CNFs). BDDs are manipulated through the CUDD package,[9] which is also used in other model checkers. The representation of Boolean functions is decoupled from the SAT checking of these functions. SAT checking in BDDs is accomplished in constant time. Thus, if CUDD is used, it has to be used for both (representation and SAT checking). The alternative representation, CNFs, relies on a self-written data structure. CNFs were mostly used during the early phases of development. A CNF quickly grows out of proportion, since there is (as of now) almost no simplification. For SAT checking of CNFs we use MiniSat,[10] but any other SAT checker could be used as well. A challenge for SAT checking is that many checks have to be executed against the FD. The CNF representation of the FD is likely to be larger than the CNF being checked against it. To avoid having to load the CNF of the FD into the SAT solver each time, we use MiniSat's ability to check satisfiability under an assumption (a literal). Basically, the CNF of the FD is loaded once. For each CNF checked against it, a temporary variable is created which is appended as a literal to each clause of the CNF. The result is then checked under the assumption that the literal is false. After this, a new clause with the temporary variable as a single negative literal is added, which corresponds to removing the clauses added before. The SAT solver is reinitialised after a number of properties have been checked (a constant, currently set to 1,000), to keep the number of temporary variables reasonably low.

[6] http://www.lsv.ens-cachan.fr/~gastin/ltl2ba.

[7] http://www.info.fundp.ac.be/~acs/tvl.

[8] http://www.cl.cam.ac.uk/~cwc22/hashtable.

[9] http://vlsi.colorado.edu/~fabio/CUDD.

[10] http://minisat.se.

5.3 Implementing the model checking algorithms

Following this overview of SNIP's architecture, we discuss some of the implementation details, and relate them to the theoretical results of [13,14] that were summarized in Sect. 3.

To conduct model checking, SNIP simulates the execution of the fPromela model. This means that (i) the calculation of the parallel composition of the processes, (ii) the calculation of the synchronous product of the processes and the never claim, and (iii) the generation of the resulting FTS according to Definition 10; are all conducted on the fly, i.e., on a per-need basis as the model checking algorithm is executed.

The model checking algorithm itself follows the `Reachables` procedure from [14] very closely. For simplicity, the procedure is implemented twice. Once as a nested DFS, which is used when an LTL property was specified (even if the LTL property is a reachability property, and thus the inner DFS is never started). In this implementation, the synchronous product with the Büchi automaton has to be calculated. This is not required when no LTL property is specified, which is why we implemented a simple DFS separately, which is used when no LTL property was specified. Checking of assertions and deadlocks is done in both cases and cannot be disabled (except for the `-fdlc` parameter discussed before) as there would be no noticeable speed gain.

5.3.1 Optimisations

There are two alternatives for making sure that only valid products are considered. One possibility is to seed the initial states with the Boolean function corresponding to the FD, the other is to test for each state whether its feature expression represents at least one valid product. In SNIP we use the latter: each time a new state is created, its feature expression is intersected with the BDD of the FD; if the intersection is empty, it is rejected. This has a number of advantages over the *seeding* method. First, with the *seeding* method, the feature expression characterising the violating products that is returned as part of the output will also contain the

Boolean function encoding of the FD, rendering it useless to the engineer. Second, the *seeding* method needs a data structure that exploits overlap in several instances. Basically, when the feature expression of all states contains the Boolean function equivalent of the FD, there will be a lot of redundancy. If the data structure used to represent feature expressions does not exploit this overlap to reduce the overall memory requirements, it will not scale. This would preclude using CNFs in this case. The CUDD package, however, does exploit overlap. After conducting experiments with both methods, though, we could not observe a noticeable difference in performance. We thus dropped the *seeding* method.

An optimisation for the ExtMc algorithm is to maintain a Boolean function characterising all violating products encountered so far, and avoiding these products in the search. SNIP has to maintain such a Boolean function already to be able to produce the summary information printed when the extended model check ends. The optimisation itself is combined with the check whether a state is reachable in valid products. As we said in the previous paragraph, the feature expression of each new state is intersected with the BDD of the FD to make sure that it contains at least one valid product. To implement the optimisation for the ExtMc algorithm, we exclude all violated products from the BDD of the FD. This way, the check required for the optimisation is conducted automatically when a new state is created, i.e., one BDD intersection instead of two.

5.3.2 Reducing fLTL to LTL

In [14], we show that fLTL model checking of an FTS (and hence of an fPromela model) can be reduced to LTL model checking of the same model with a modified FD. SNIP uses this result, which consists in appending the fLTL guard (specified with the `-filter` parameter) as a constraint to the TVL model, thus insuring that only products at the intersection of the FD and the guard are considered. This is done before the TVL library is called, which means that SNIP does not even have to parse the quantifier. Since quantifiers are specified with a separate parameter, they can not only be used for LTL properties, but also for checking assertions and deadlocks.

5.3.3 Overview

In summary, when SNIP model checks an fLTL property $[\chi]\phi$, it proceeds as follows. The initialisation consists of three steps. First, SNIP translates the LTL property ϕ to a Büchi automaton and appends it to the fPromela file as a never claim. Second, SNIP appends the quantifier χ as a constraint to the TVL model and transforms it into a BDD. Third, SNIP parses the fPromela file, creating one or more featured program graphs in the process. After the initialisation, SNIP launches the model checking algorithm. The algorithm computes the FTS corresponding to the parallel composition of the featured program graphs and their synchronous product with the never claim. It uses a depth-first search to compute the reachable states (stored in a hash table) and for each, the products in which it is reachable (in the form of a BDD). For each new state, SNIP makes sure that it is reachable in a valid product which is not yet known to violate the property. When a violating state is found, SNIP prints information about the violation (the feature expression characterising the violating products is obtained from the BDD, and a counterexample). If the `-exhaustive` parameter is set, SNIP continues the search and prints a summary of all violations when the algorithm finishes.

5.3.4 Projection

Finally, we would like to point out that SNIP has a parameter `-spin`, which causes it to interpret any input model as a Promela file. This means that feature variables are treated like normal Boolean variables, and that `gd` statements are treated like `if`s. The input will thus be interpreted as a featured program graph without feature expressions, i.e., a normal program graph. In other words, the use of this parameter results in applying the syntactic projection described in Algorithm 2. Because feature variables are considered as Boolean variables, they all have an initial value (which is 0 if no explicit value is given). Therefore, the resulting program graph corresponds to the projection of the FTS to a product p (see Theorem 1), namely the one corresponding to the value of each feature variable. In this case, no BDDs (not even trivial ones) will be computed and SNIP's model checking algorithm is equivalent to the classical model checking algorithm for single systems.

This option is intensively used for our benchmarks. Indeed, it allows us to use SNIP to compute the naïve algorithm (described in Sect. 3.3). We can then compare the naïve algorithm to the FTS algorithm where both are implemented by the same tool (even the same code). In an experiment measuring performance, this allows us to control many variables that would be impossible to control if different tools were used to perform the comparison.

6 Experiments

An initial set of experiments conducted with a Haskell FTS library [13] showed that in practice, the semi-symbolic FTS algorithm is up to three times faster than the naïve algorithm. There were some limitations to this evaluation, which we overcome here and in [14]: it only considered a limited number of properties (six), a rather small model (457 states in the FTS), and it did not measure the state space reduction.

We thus conducted new experiments with SNIP. For doing so, we considered two case studies. The first case study is the

mine pump system [29] discussed in Sect. 5.1. It has 11 features and 128 products; its FTS has 21,177 states, all products combined leads to a TS of 889,252 states. The second model represents a subset of the CCSDS file delivery protocol (CFDP) [15], with 10 features and 56 products; its FTS has 1,064,840 states, and the sum of all products combined leads to a TS of 2,780,475 states. The results of the mine pump experiments can be found in [14]. Here, we report on those of the CFDP experiments.

Each of the two case studies required the creation of both an fPromela model and a TVL model. The translation of the feature diagrams into TVL models was straightforward. Creating the fPromela models required a similar effort to creating normal Promela models. Most of the complexity is due to the fact that such formal models are very delicate, that is, it is not always clear how a change in the model will impact a property. Moreover, this task was complicated by the fact that our models had to follow an existing specification. Through these modelling tasks, we tried to assess the added complexity due to the new concept of *feature* in fPromela. We found that it did not add to the complexity of Promela in any significant way, and that it integrated rather well with the existing concepts.

The fPromela models, including all properties and explanations, are distributed with SNIP [9]. The full set of results is also available online.[11]

6.1 Experimental setup

Our experimental setup consists of SNIP and a script that implements the naïve algorithms using SPIN and SNIP without the FTS algorithm (referred to as 'enum (snip)' in the statistics). The script uses the TVL library to list the set of valid products. For each, it transforms the fPromela input into a Promela file that describes the behaviour of the product (following Algorithm 2). It then uses first SNIP without the FTS algorithm, then SPIN, to model check the file. While the script makes up for the lack of functionality in SPIN (and SNIP without the FTS algorithm), it is still inferior in terms of usability. For instance, SNIP produces a Boolean expression characterising the violating products. As shown in Sect. 5.1, this expression identifies incompatible features, or features that are required for a property to hold. The script, in contrast, only lists the products that violate the property. The list has to be analysed again to produce information comparable to that returned by SNIP.

SNIP without the FTS algorithm provides a baseline to evaluate the impact of the FTS algorithm on runtime and size of the state space. A meaningful evaluation of the runtime cannot be done by comparison to tools such as SPIN, as it would require us to remove the bias introduced by optimisations for single systems. The relevant comparison is thus between SNIP with and without the FTS algorithm. However, SPIN can be used to evaluate the ability of our algorithm to reduce the state space.

We only consider ExtMc in our experiments. The performance of the naïve Mc algorithm largely depends on the order in which products are checked, which we want to exclude as a factor.

Our experiments consist in using SNIP and the above script to compute ExtMc for all properties of the two models. For each, we measured the runtime and the number of explored states. Recall that the FTS algorithm can re-explore states. The sum of the explored and re-explored states corresponds to the number of transitions fired. In the case of the naïve algorithm, this number is equal to the number of explored states. Henceforth, we will thus use 'number of transitions' rather than 'sum of explored and re-explored states'. To make measurements as fair as possible, the time counted for the naïve algorithm only includes the verification time. Moreover, in the case of SPIN, verification consists of three steps: (a) generating a process analyser (pan), (b) compiling it and (c) running it. The time for (b) was not counted, as it is due to a design decision in SPIN rather than its model checking algorithm. All benchmarks were run on an Ubuntu machine with an Intel Core2 Duo at 2.80 GHz with 4 Gb of RAM.

6.2 CFDP

The *Consultative Committee for Space Data Systems* (CCSDS)[12] is an international organisation founded by several space agencies. Its aim is to develop communication and data system standards for spatial missions. One of those standards is the *CCSDS File Delivery Protocol* (CFDP) [15], a communication protocol intended for deep-space file transfer between several spacecrafts and ground stations. The protocol also includes the possibility to perform operations on the storage medium.

The CFDP is highly configurable, and is thus suitable for a wide variety of missions. For example, a mission may require a given (negative) acknowledgment mode in order to ensure that files are transmitted successfully. In another context, acknowledgements may be useless or even unsuitable, e.g., when the transmission must be completed as fast as possible. Furthermore, the CFDP is also compatible with a wide range of subnetwork services. A mission usually only needs a subset of its functionality. To minimise the memory requirements of the CFDP, the non-required parts are not implemented. As part of a collaboration with Spacebel, a Belgian company that develops software for space missions, the CFDP specification was analysed and the protocol decomposed into features. This feature decomposition was subsequently used in

[11] http://www.info.fundp.ac.be/~acs/snip/benchmarks.

[12] http://www.ccsds.org.

the development of a CFDP library SPL [6]. We used it as the basis for our CFDP models, in which we consider a small subset of the protocol.

At the heart of the protocol is the transmission of files between CFDP entities, that is, spacecrafts and ground stations. A transmission starts with the sender transferring a metadata segment to the receiver, followed by data segments composing the file to be transmitted. Once all data segments are transmitted, the sending entity sends an *End-Of-File* (EOF) message and the receiver closes the transaction by sending a *Finished* (FIN) message.

Our experiment considers the efficiency of the different *Negative Acknowledgement* (NAK) procedures offered by the CFDP to detect and retransmit lost data segments. The protocol provides four NAK modes:

Deferred. The receiving entity waits until the EOF message before it requests missing data segments.

Immediate. The receiving entity requests a missing data segment as soon as it notices the loss.

Prompted. At any point during the transmission, the sender can prompt the receiver (using a PROMPT message) to request the retransmission of lost data segments. In addition, the receiver will request all missing data segments when the EOF message is received (as in *deferred* mode).

Asynchronous. At any point during the transmission, the receiver can request the retransmission of data segments lost up to this point.

Because we are concerned only with the variability in the NAK modes, we consider only a small subset of the FD created for [6]. In TVL, this subset is the following.

```
root CFDP {                                  1
  group allOf {                              2
    Entity group [0..*] {                    3
      Snd_min group [0..*] {                 4
        Snd_min_ack group [0..*] {           5
          Snd_prompt_nak                     6
        }                                    7
      },                                     8
      Recv_min group [0..*] {                9
        Recv_min_ack group [0..1] {          10
          Recv_immediate_nak,                11
          Recv_deferred_nak,                 12
          Recv_prompt_nak,                   13
          Recv_asynch_nak                    14
        }                                    15
      }                                      16
    },                                       17
    Channel group [0..*] {                   18
      Reliable                               19
    }                                        20
  }                                          21
}                                            22
```

Note that the FD also models the reliability of the communication channel. Even though this is strictly speaking a property of the environment, not of the system, it is useful to capture it as as a feature. The truth of any property is then automatically expressed in function of the reliability of the communication channel. It is also worth mentioning that the features corresponding to the four NAK modes are mutually exclusive. Consequently, the FD has 56 products.

The fPromela model of the CFDP represents the scenario in which a file is transmitted between two CFDP entities. With this model we verify under which conditions (i.e., with which features) the file will be successfully transmitted to the receiving entity. The model is based on the CFDP specification [15], rather than code developed by Spacebel. Because we are only interested in the transmission procedure, the CFDP operations that are unrelated to the transmission itself (user requests, checksum errors,...) are ignored. Moreover, we applied some simplifications to the transmission procedure as described in the protocol specification.

For the benchmarks, we considered a deadlock check, property (#1), and the following fLTL properties.

(#2) The whole file is eventually received, $\Diamond\, fileReceived$. This property is violated by 38 products, all those where the communication channel is not reliable, and those without the sending or without the receiving feature.

(#3) If the EOF message eventually reaches the receiver, the whole file is eventually received, $\Diamond eofReceived \Rightarrow \Diamond\, fileReceived$. This property is violated 18 products: all those where the channel is not reliable and with both a sender and a receiver (otherwise the assumption would not hold).

(#4) The same as (#3) with the additional assumption that a negative acknowledgement (NAK) eventually reaches the sending entity:

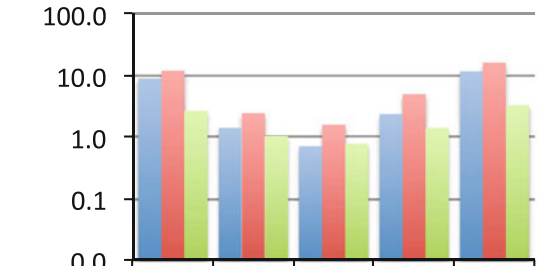
$$(\Diamond eofReceived \wedge \Diamond nakReceived) \Rightarrow \Diamond\, fileReceived.$$

This property is violated by 9 products, those with an unreliable channel and where either

- the receiver is in asynchronous, immediate, or prompted NAK mode,
- or the receiver is in deferred NAK mode but the sender is unable to answer the NAK messages.

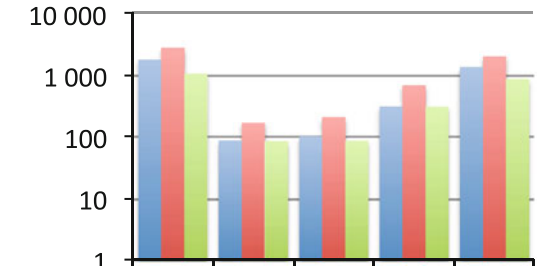
(#5) A variation of the previous property where the second assumption is that the sender receives NAK messages infinitely often.

$$(\Diamond eofReceived \wedge \Box\Diamond nakReceived) \Rightarrow \Diamond\, fileReceived.$$



	#1	#2	#3	#4	#5
snip	8.86	1.40	0.71	2.33	11.48
enum (snip)	11.75	2.42	1.58	4.91	15.93
enum (spin)	2.64	1.04	0.77	1.39	3.28

(a)



	#1	#2	#3	#4	#5
snip	1 802	88	103	310	1 359
enum (snip)	2 780	169	210	687	2 017
enum (spin)	1 063	85	86	309	859

(b)

Fig. 6 Benchmark results for the CFDP. **a** Runtime in seconds, logarithmic scale. **b** Number of transitions in thousands, logarithmic scale

> This property is violated by 4 products: those where the communication channel is unreliable, the receiver has enabled a NAK mode and the sender is unable to answer NAK messages.

The results for the aforementioned properties are presented in Fig. 6. In terms of runtime, the FTS algorithm in SNIP is between 1.33 and 2.23 times faster than the naïve algorithm implemented with SNIP.

In terms of state space, the FTS algorithm in SNIP reduced the average number of states from 1,440,675 to 910,497 (37 %), whereas SPIN reduced it to 579,077 (60 %). The small reduction in the state space explains the lesser performance of the FTS algorithm in this case. Indeed, we observe that the greater this reduction, the larger the difference in runtime between the two algorithms. The significant reduction in the case of SPIN is most likely due to optimisations such as partial order reduction. The model in question is a distributed system in which partial order reductions can lead to significant reductions in the size of the state space.

As to memory consumption, the deadlock check required the most: 336.9 MB for the naïve algorithm and 395.4 MB for the FTS algorithm. These sizes correspond to 933,276 states in the TSs explored by the naïve algorithm and 1,069,840 states in the FTS. Here, the FTS algorithm requires 17 % more memory, for a 14 % increase in the number of states. Again, there is 3 % overhead for the BDDs representing sets of products.

6.3 Incremental benchmarks

While the previous experiments compared the FTS algorithm and the naïve algorithm on a fixed number of products, we also conducted an experiment to evaluate how each algorithm behaves when the number of products increases. For this we used the CFDP model of the previous section and property (#1), the deadlock check. We first verified the model restricted to 18 products, with the following five features: *Snd_min*, *Snd_min_ack*, *Recv_min*, *Recv_min_ack*, and *Reliable*. We then reverified the model five times, each time adding one feature in the following order: *Recv_immediate_nak* (24 products), *Recv_deferred_nak* (30 products), *Recv_asynch_nak* (36 products), *Snd_prompt_nak* (48 products), and *Recv_prompt_nak* (56 products).

For both runtime and state space, the increase when adding the sixth feature, *Recv_immediate_nak*, is huge. The reason for this increase is that in the *immediate* mode, the receiver sends a NAK as soon as a loss is noticed. This NAK itself can get lost, which leads to a large number of combinations for the lost/received data segments and lost/received NAK messages. Without the *Recv_immediate_nak* feature, the FTS has 16,801 states and the TSs of all products combined have 98,112, i.e., the FTS is 78 % smaller. When the feature is added, the FTS grows to 917,066 states (by 5,300 %) and the TSs altogether to 1,192,023 states (by 1,114 %). Now the FTS is only 15 % smaller. Basically, the states added by the *Recv_immediate_nak* feature have a negative impact on the compactness of the FTS.

For the other features, the increase for the FTS algorithm is consistently lower than for the naïve algorithm. For example, when the number of features increases from eight to nine, the number of products increases from 36 to 48 and the runtime of the FTS algorithm grows only by 57 %, while the runtime of the naïve algorithm rises by 85 % when implemented with SNIP and by 72 % when implemented with SPIN.

We conducted these incremental benchmarks also for the other properties and other orders of features, observing similar results. As the number of features increases, the runtime for the FTS algorithm grows slower than that of the naïve algorithm. This indicates that the FTS algorithm scales better with the number of features than the naïve algorithm.

This can be visualised by comparing the rate at which the runtime increases with the rate at which the number of prod-

ucts increases. For the algorithm to scale with the number of features, its runtime should increase linearly in the number of features (i.e., logarithmically in the number of products), rather than exponentially. To test this, we take the runtime for checking the model consisting of 24 products as the baseline, and extrapolate the runtime for the models of 30, 36, 48 and 56 products in two ways.

Exponential. The runtime increases at the same rate as the number of products, i.e., by 25% for the increase to 30, by 50% for the increase to 36 and so on. This growth is exponential in the number of features.

Linear. The runtime increases at the same rate as log_2 of the number of products, i.e., by 7% for the increase to 30, by 13% for the increase to 36 and so on. This growth is linear in the number of features.

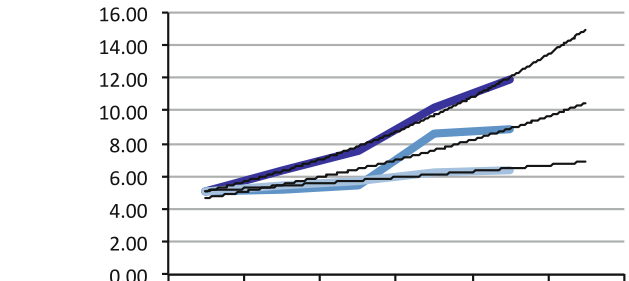
As the baseline we chose the runtime for the model of 24 products, because the runtime for the model of 18 products (i.e., without the *Recv_immediate_nak* feature) is an outlier for all three algorithms.

The result is shown in Fig. 7, where we plot the projected runtimes as well as the measured runtime for each algorithm and implementation. To make the results easier to interpret, we further added a function that approximates each line (in black). As can be seen clearly in these figures, the runtime of the FTS algorithm increases at a rate that is between exponential and linear, whereas that of the naïve algorithms increases at the exponential rate (whether implemented with SNIP or with SPIN).

6.4 Discussion

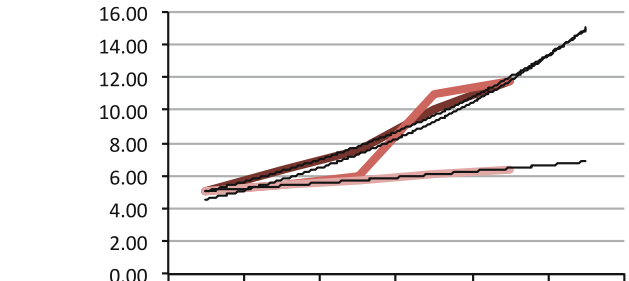
These results (and those in [14]) show that the FTS algorithm is a viable approach for state space reduction, in some cases leading to better performance than the script using SPIN, a tool that has been under development for over 30 years. There is, nevertheless, room for improvement of the implementation. Furthermore, we believe that the state space reductions of SNIP and SPIN can reinforce each other, opening an exciting area of future work. One step will be to extend the optimisations currently implemented in SPIN, such as partial order reduction, to FTS. The FTS algorithm could then be integrated into SPIN. While such a project will most likely be more expensive than the development of SNIP, its prospects are promising.

As for the threats to validity, all experiments were executed by an automated script running on a dedicated machine. This minimises the risks of flawed runtime measures, e.g., due to other processes. Although the tools were the only running processes on the machine, all benchmarks were run five times (and the runtimes averaged) to avoid the influence of independent random variations. Furthermore, the runtime



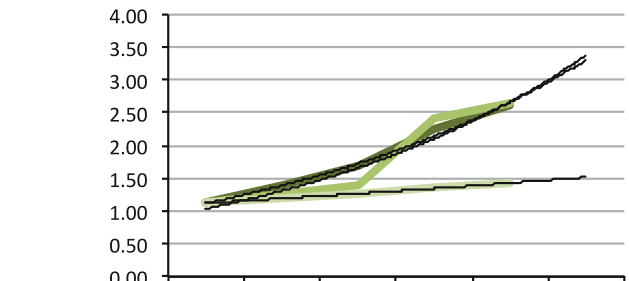
	6 (24)	7 (30)	8 (36)	9 (48)	10 (56)	
Exponential	5.08	6.35	7.62	10.16	11.85	
Measured	5.08	5.24	5.51	8.66	8.86	
Linear	5.08	5.44	5.73	6.19	6.43	

(a)



	6 (24)	7 (30)	8 (36)	9 (48)	10 (56)	
Exponential	5.06	6.33	7.59	10.12	11.81	
Measured	5.06	5.43	5.95	11.01	11.75	
Linear	5.06	5.42	5.71	6.16	6.41	

(b)



	6 (24)	7 (30)	8 (36)	9 (48)	10 (56)	
Exponential	1.12	1.40	1.68	2.24	2.61	
Measured	1.12	1.26	1.41	2.42	2.64	
Linear	1.12	1.20	1.26	1.36	1.42	

(c)

Fig. 7 Growth of the runtime with increasing number of features. **a** FTS algorithm with SNIP. **b** Naïve algorithm with SNIP. **c** Naïve algorithm with SPIN

reported for the naïve algorithm is only the verification time. This removes the bias that might have been caused by an inefficient implementation of the script iterating through the products, generating the input files, deleting them, and so forth.

From the obtained results it is clear that the FTS algorithm is in most cases an improvement over the naïve algorithm, both in terms of runtime and in terms of state space. However, the use a single model means that this is a quasi-experiment and that the extent to which it generalises cannot be concluded from its results. Nevertheless, we used several properties, including both liveness and safety (mostly combinations of both) and covering properties satisfied under various circumstances (i.e., violated by different sets of products).

7 Conclusion

We presented SNIP, a tool that tackles the SPL model checking problem. It combines an implementation of the semi-symbolic FTS algorithms with two high-level specification languages. Its modelling language, fPromela, is an extension of Promela, the language used by the popular model checker SPIN. SNIP also uses TVL, a text-based language for defining the features of an SPL and the constraints between them.

SNIP was implemented from scratch. Although this was a time consuming and risky undertaking, it has given us many insights into the use of the semi-symbolic FTS algorithm as part of a model checker for a non-trivial language. Experiments conducted with SNIP have shown that the FTS algorithm is generally faster than the naïve algorithm, and a viable approach for state space reduction. Furthermore, from the experiment results, we expect the gap between the two algorithms to increase as the number of features in a given SPL gets larger. However, another performance evaluation with a higher number of features should be carried out before drawing conclusions.

In spite of these positive results, we are aware that there is still much room for improvement. Indeed, the model checker SPIN, which has been under development for more than 30 years, generally outperforms our tool thanks to many optimisations, such as partial order reduction or statement merging. This opens both theoretical and practical challenges, because these optimisations have not been studied for FTS yet. In future work, we plan to improve the efficiency of the FTS model checking. In this regard, we notably aim to (1) combine our fLTL verification algorithm with distributed LTL model checking; (2) define abstraction relations (e.g. bisimulation and simulation) for an FTS in order to exploit them to verify a more abstract FTS; (3) study how to apply optimisations such as partial order reduction.

Acknowledgments This work was funded by the FNRS, the Interuniversity Attraction Poles Programme of the Belgian State, Belgian Science Policy (MoVES project) and the BNB.

References

1. Apel, S., Speidel, H., Wendler, P., von Rhein, A., Beyer, D.: Detection of feature interactions using feature-aware verification. In: ASE 2011, pp. 372–375. IEEE (2011)
2. Asirelli, P., ter Beek, M.H., Fantechi, A., Gnesi, S.: A logical framework to deal with variability. In: IFM '10. LNCS, vol. 6396, pp. 43–58. Springer, Berlin (2010)
3. Asirelli, P., ter Beek, M.H., Fantechi, A., Gnesi, S.: Formal description of variability in product families. In: SPLC'11, pp. 130–139. IEEE CS (2011)
4. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press, Boca Raton (2007)
5. Batory, D.S.: Feature models, grammars, and propositional formulas. In: SPLC '05. LNCS, vol. 3714, pp. 7–20. Springer, Berlin (2005)
6. Boucher, Q., Classen, A., Heymans, P., Bourdoux, A., Demonceau, L.: Tag and prune: a pragmatic approach to software product line implementation. In: ASE '10, pp. 333–336. ACM, New York (2010)
7. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv. **24**(3), 293–318 (1992)
8. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: a new symbolic model checker. Int. J. Softw. Tools Technol. Transf. **2**, 410–425 (2000)
9. Classen, A. http://www.info.fundp.ac.be/~acs/fts. (2010)
10. Classen, A.: Modelling and Model Checking Variability-Intensive Systems. PhD thesis, PReCISE Research Centre, Faculty of Computer Science, University of Namur (FUNDP), 5000 Namur, Belgium (2011)
11. Classen, A., Boucher, Q., Heymans, P.: A text-based approach to feature modelling: syntax and semantics of TVL. Sci. Comput. Programm. **76**, 1130–1143 (2011)
12. Classen, A., Heymans, P., Schobbens, P.-Y., Legay, A.: Symbolic model checking of software product lines. In: ICSE '11, pp. 321–330. ACM, New York (2011)
13. Classen, A., Heymans, P., Schobbens, P.-Y., Legay, A., Raskin, J.-F.: Model checking lots of systems: Efficient verification of temporal properties in software product lines. In: ICSE '10, pp. 335–344. ACM, New York (2010)
14. Classen, A., Heymans, P., Schobbens, P.-Y., Legay, A., Raskin, J.-F.: Modelling and model checking variability-intensive systems with featured transition systems. IEEE Trans. Softw. Eng. (2012) (Submitted)
15. Consultative Committee for Space Data Systems (CCSDS). *CCSDS File Delivery Protocol (CFDP): Blue Book, Issue 4*. Number CCSDS 727.0-B-4. NASA (2007)
16. Czarnecki, K., Antkiewicz, M.: Mapping features to models: a template approach based on superimposed variants. In: Gluck, R., Lowry, M. (eds.) GPCE '05. LNCS, vol. 3676, pp. 422–437. Springer, Berlin (2005)
17. Fantechi, A., Gnesi, S.: Formal modeling for product families engineering. In: SPLC '08, pp. 193–202. IEEE (2008)
18. Fischbein, D., Uchitel, S., Braberman, V.: A foundation for behavioural conformance in software product line architectures. In ROSATEA '06, ISSTA '06 workshop, pp. 39–48. ACM, New York (2006)
19. Garrido, A., Johnson, R.: Analyzing multiple configurations of a C program. In: ICSM '05, pp. 379–388. IEEE (2005)

20. Gastin, P., Oddoux, D.: Fast LTL to Bnchi automata translation. In: CAV '01. LNCS, vol. 2102, pp. 53–65. Springer, Berlin (2001)
21. Gruler, A., Leucker, M., Scheidemann, K.: Modeling and model checking software product lines. In: FMOODS '08. LNCS, vol. 5051, pp. 113–131. Springer, Berlin (2008)
22. Holzmann, G.J.: State compression in SPIN. In: the 3rd SPIN Workshop (1997)
23. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, Menlo Park (2004)
24. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. **41**(4), 21:1–21:54 (2009)
25. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, SEI (1990)
26. Kästner, C., Apel, S., Kuhlemann, M.: Granularity in software product lines. In: ICSE '08, pp. 311–320. ACM, New York (2008)
27. Kästner, C., Giarrusso, P.G., Ostermann, K.: Partial preprocessing C code for variability analysis. In: VaMoS '11, ICPS, pp. 127–136. ACM, New York (2011)
28. Kishi, T., Noda, N.: Formal verification and software product lines. Commun. ACM **49**(12), 73–77 (2006)
29. Kramer, J., Magee, J., Sloman, M., Lister, A.: CONIC: an integrated approach to distributed computer control systems. IEEE Proc. Comput. Digit. Tech. **130**(1), 1–10 (1983)
30. Larsen, K.G.: Modal specifications. In: Automatic Verification Methods for Finite State Systems. LNCS, vol. 407, pp. 232–246. Springer, Berlin (1989)
31. Larsen, K.G., Nyman, U., Wasowski, A.: Modal I/O automata for interface and product line theories. In: ESOP '07. LNCS, vol. 4021, pp. 64–79. Springer, Berlin (2007)
32. Lauenroth, K., Töhning, S., Pohl, K.: Model checking of domain artifacts in product line engineering. In: ASE '09, pp. 269–280. IEEE/ACM (2009)
33. Liu, J., Dehlinger, J., Lutz, R.: Safety analysis of software product lines using state-based modeling. J. Syst. Softw. **80**(11), 1879–1892 (2007)
34. Plath, M., Ryan, M.: Feature integration using a feature construct. Sci. Comput. Program. **41**(1), 53–84 (2001)
35. Pnueli, A.: The temporal logic of programs. In: FOCS '97, pp. 46–57. IEEE (1977)
36. Post, H., Sinz, C.: Configuration lifting: Verification meets software configuration. In: ASE'08, pp. 347–350. IEEE CS (2008)
37. Schobbens, P.-Y., Heymans, P., Trigaux, J.-C., Bontemps, Y.: Feature Diagrams: A Survey and A Formal Semantics. In: RE '06, pp. 139–148. IEEE CS (2006)
38. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: LICS '86, pp. 332–344. IEEE (1986)
39. Ziadi, T., Hélouët, L., Jézéquel, J.-M.: Towards a UML profile for software product lines. In: van der Linden, F. (ed.) PFE '03. LNCS, vol. 3014, pp. 129–139. Springer, Berlin (2003)