SNIP: An Efficient Model Checker for Software Product Lines

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

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

TECHNICAL REPORT July 28, 2011

AUTHORS	A. Classen, P. Heymans, M. Cordy, P.-Y. Schobbens, A. Legay
APPROVED BY	P. Heymans
EMAILS	{acs}@info.fundp.ac.be
STATUS	Draft.
REFERENCE	P-CS-TR SPLMC-00000003
PROJECT	MoVES
FUNDING	FNRS, the Walloon Region, Interuniversity Attraction Poles Programme of the Belgian State of Belgian Science Policy

SNIP: An Efficient Model Checker for Software Product Lines

SNIP: An Efficient Model Checker for Software Product Lines

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

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

ABSTRACT

In software product lines, systems are developed in families and differences between systems of a product line are expressed in terms of features. The model checking problem for product lines is more difficult than for single systems because a product line with n features yields up to 2^n individual systems to verify. This paper introduces SNIP, a tool for model checking product lines against temporal properties. SNIP is the first model checker for software product lines. It relies on an efficient mathematical structure for product line behaviour, that exploits similarities and represents the behaviour of all systems in a compact manner. This structure is used to model check all systems of the product line in a single step. The tool comes together with an intuitive specification language based on Promela. We compare SNIP to classical model checkers.

1. INTRODUCTION

Software Product Lines (SPLs) are a popular software engineering paradigm that seeks to maximise reuse by planning upfront which *features* should be common, resp. variable, for several similar software systems [7] (called *products*).

SPLs are used for the development of embedded and critical systems [9]. Approaches for formal modelling and model checking of SPL behaviour are thus vital for quality assurance and are actively studied. The *model checking* problem consists in checking whether a system model satisfies a given *temporal logic* property [1]. The model checking problem in SPLs is different from the one in single systems engineering. To conduct analysis on the level of the SPL (as opposed to analysing individual products), the model and the algorithm have to take variability into account. The algorithm has to consider all products as part of the check, and pinpoint those that violate it [6]. This means that the modelling language has to offer a way to specify the behaviour of all products concisely, while still retaining traceability from behaviours to products. The model checking problem is also harder, as it has to deal with the fact that there can be exponentially many, $O(2^{\#features})$, products to verify.

In [5, 4], we addressed the model checking problem for SPLs and *Linear Temporal Logic* (LTL) by introducing *Featured Transition Systems (FTS)*, a mathematical formalism to express the behaviour of all products of the SPL in one model. FTS are transition systems in which transitions are labelled with *feature expressions* (in addition to being labelled with actions). This allows one to keep track of the different products. We also proposed a semi-symbolic model checking algorithm [5] that exploits the structure of the FTS and tries to avoid an exponential number of verifications.

Here we introduce SNIP, a model checker for SPLs that implements the algorithms of [5]. To the best of our knowledge, SNIP is the first model checker specifically dedicated to SPLs. We also propose a specification language, fPromela (based on Promela [11]), that allows to specify SPLs concisely and intuitively. Our experiments show that SNIP performs faster than model checking of individual products.

SNIP is available at www.info.fundp.ac.be/~acs/snip. It is distributed in source code form and comes with documentation and case studies.

2. SNIP: AN OVERVIEW

The principal difference between SPL model checking and single systems model checking is the presence of *variability*. Variability in SPLs is typically expressed in terms of *features*, that is, a product of the SPL is specified as a set of features. Since features serve as the central unit of difference in SPLs, it is imperative that model checking approaches recognise them as a first-class concept, that is, inputs and results should be expressed in terms of features. There are, however, currently no model checkers that do this, except for SNIP and one of our earlier model checkers [4]. Furthermore, *feature diagrams* are commonly used to capture known dependency or exclusiveness relations of features (they specify the set of *legal* products [12, 18]). They should also be part of a model checking approach. Otherwise, the model checker might identify problems in products that are not legal in the first place. Currently only SNIP has this capability.

Let us describe how SNIP addresses these requirements with its modelling language and user interface.

2.1 The fPromela modelling language

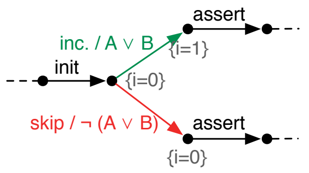
In SNIP, SPLs are specified with *fPromela*, that is, an extension of the well-known *Promela* language of SPIN [11]. fPromela extends Promela with a new type, *feature variables*. These can be used to *guard* statements with *feature*

[*]FNRS Research Fellow

.

```
active proctype toto() {
    ...
    int i = 0;
    gd :: f.A || f.B -> i++;
       :: else       -> skip;
       dg;
    assert(i == 1);
    ...
}
```

(a) An fPromela fragment.



(b) The corresponding FTS.

Figure 1: Example of fPromela and its semantics.

expressions. A guarded statement is part of the model of a product if its guard evaluates to *true* in the product. An example of code written in fPromela is given in Figure 1(a), where the increment statement is guarded by the feature expression `f.A` $\vee$ `f.B`. This means that `i` is only incremented in products containing features `f.A` or `f.B`. In fPromela, any statement can be guarded and guards can be nested. fPromela includes almost all constructs of Promela.

Given an fPromela model, the behaviour of a product is obtained by fixing the values of all feature variables.

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

In addition to fPromela, the TVL feature modelling language [3] is used to declare features and constraints between them. Our model checking algorithm exploits this information in order to speed up computation and avoid exploring states that do not belong to any legal product.

Our choice of Promela as the basis of SNIP's input language is motivated by its widespread use and relative ease of specification. Furthermore, our choice to model variability with guards is motivated by the widespread use of similar techniques in practice (e.g., ifdefs [13, 17], code tags [2] or coloured annotations [13]). In this regard, it differs from our earlier language [4], where features are specified as modules and weaved into a base system. The use of guards is more intuitive, especially due to its straightforward semantics. This makes sure that fPromela's learning curve is rather gentle.

2.2 User interaction

Having features as a first-class concept means that results of a model checker have to be provided in terms of features. This is not the case if just a single product is verified, or a list of products one by one. Either case yields information about specific products, which is limiting as problematic features cannot be inferred from violating products. This is not only limited as a verification result, but also inappropriate for the engineer who thinks in terms of features when specifying the model. Without knowing which features are responsible, it is much more difficult to locate an error, especially if it involves several interacting features. Extending classical model checkers to SPLs almost inevitably leads to this situation. SNIP and our earlier model checker [4] are the only tools that present their results in terms of features.

In SPL model checking, one distinguishes two model checking problems [5]. Assume that a property ϕ is violated by one or more products: $Mc(\phi)$ returns $false$ and identifies at least one product while $ExtMc(\phi)$ returns $false$ and identifies *all* violating products. This already yields two use cases for a model checker. Furthermore, it is sometimes necessary to verify properties that are only relevant for products containing certain features (e.g., if they correspond to a requirement implemented by a feature). This combined with the above problems leads to four use cases. SNIP is currently the only model checker supporting all of them.

The user interface of SNIP was designed to take these use cases into account. It also addresses a variety of practical concerns that the user might have, like simulation, bounded checking, layout of stack traces, and so on. As typical for model checkers, the user interface is command-line based. To cover the four use cases explained above, SNIP has two parameters. Normally, SNIP halts its execution once a violation is found. This corresponds to $Mc(\phi)$, as the full set of products that violate the property may not have been computed yet. To force SNIP to compute the full set, i.e., $ExtMc(\phi)$, one can use the parameter `-exhaustive`. SNIP will then continue the search until either all products are found to violate the property or until the full state space is explored. The user can also restrict the verification to a subset of products using the `-filter` parameter. The subset is specified as a feature expression in TVL syntax.

Properties for SNIP can be specified in LTL or as assertions (SNIP also detects deadlocks). In Figure 1(a), the property that i equals 1 is specified with an assertion. For the above example, a call to SNIP would look as follows:[1]

```
$ ./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):
   (!A & !B)
 - Stack trace:
   ...
```

As expected, SNIP reports that the assertion is violated by all products in $[\![\neg$`f.A`$\wedge\neg$`f.B`$]\!]$. If the feature model is changed in order to impose the presence `f.A` or `f.B`, SNIP will report that the property is satisfied.

To be able to correct an error, the model checker usually provides a counterexample. In the case of SPL model checking, the counterexample has to correspond to the violating products that were returned. Especially in the case of $ExtMc$, that might mean that the model checker has to return several counterexamples, each corresponding to a subset of the violating products. Here also, SNIP is currently the only model checker that does this.

What we just described are four decision problems and a high-level language for FTS, the underlying formalism [5]. We now discuss how SNIP solves these problems.

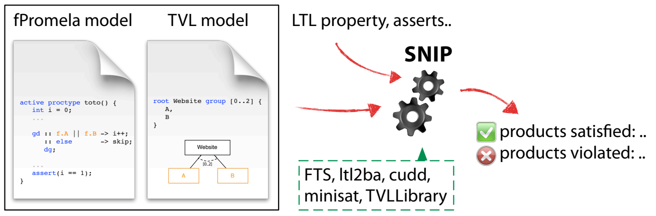
3. UNDER THE HOOD

Now that we have given a functional overview of SNIP, we dive into more technical aspects.

3.1 Theoretical foundations

As in most model checking approaches, the behaviour of an individual product is specified by a transition system. In [5], we proposed FTS, a compact model for representing the behaviours of *all the products* of an SPL. FTSs are transition systems in which individual transitions are labelled with Boolean functions over the features, called *feature expressions*. The differences between all the transition systems in the SPL are either addition or removal of states to

[1]The `-fm` parameter specifies the feature model (can be omitted if the TVL file has the same name as the fPromela file).



```
active proctype toto() {
   int i = 0;
   ...

   gd :: f.A || f.B -> i++;
      :: else       -> skip;
      dg;

   ...
   assert(i == 1);
}
```

```
root Website group [0..2] {
   A,
   B
}
```

Figure 2: Inputs and outputs of SNIP.

a common structure shared by all the products. Labelling by feature expressions allows to capture such modifications.

Definition 2. *An FTS is a tuple $fts = (S, Act, trans, I, AP, L, d, \gamma)$, where S is a set of states, Act a set of actions, $trans \subseteq S \times Act \times S$ a transition relation, $I \subseteq S$ the set of initial states, AP a set of atomic propositions and $L : S \to 2^{AP}$ a labelling function, d is a feature model, and $\gamma : trans \to \left(\{0,1\}^{|N|} \to \{0,1\}\right)$ is a total function, labelling each transition with a* feature expression, *i.e., a Boolean function over the set of features.*

FTS provide the semantic foundation for fPromela. Basically, an fPromela model is a high-level description of an FTS. This is illustrated in Figure 1(b): after an initial transition, the variable i is incremented in the products with features A or B (green) and otherwise left untouched (red).

Verification of an FTS should produce the list of products that satisfy a property. To this end, one can easily use an enumerative approach, by verifying each product individually. In [5], we proposed a more efficient algorithm, which exploits similarities between products in order to reduce the state space. The theoretical complexity of this algorithm is the same as for the enumerative approach. However, preliminary benchmarks showed our algorithm to be faster [5].

3.2 Implementation

SNIP performs verification by on-the-fly-generation of the FTS that corresponds to the fPromela model given as input. Compared to the enumerative approach, SNIP does not generate the 2^n transition systems that can be derived from the FTS. In the enumerative approach, the feature variables are considered to be part of the system state. SNIP, in contrast, treats them symbolically: each state has a Boolean function that represents the products for which the state is reachable.

Let us illustrate this with Figure 1(b). The initial state of the FTS is reachable in all products. SNIP thus internally attaches the feature expression *true* to the state. When SNIP executes the first transition, it knows that the second state is also reachable in all products. While SNIP executes this transition once, the enumerative algorithm would have to execute it for each product. Our algorithm can thus drastically reduce the number of states that have to be visited. When SNIP fires the red transition, it learns that the target state is reachable in products $[\![\neg(A \vee B)]\!]$. It then tests the assertion, which fails, and so SNIP reports an error for products $[\![\neg(A \vee B)]\!]$. All operations involving feature expressions are computed over symbolic sets. For instance, in order to check whether a state with feature expression f is reachable in products $[\![p]\!]$, we test $[\![p]\!] \subseteq [\![f]\!]$.

A schematic summary of the inputs and outputs of SNIP is presented in Figure 2. Internally, SNIP uses LTL2BA[2] to generate never claims from LTL formulas, CUDD[3] to store Boolean functions as BDDs, MiniSat[4] as an alternative to CUDD, and the TVL library[5] to transform a TVL model into a Boolean function. SNIP is written in C. It was implemented from scratch and only shares the parser with SPIN. This was necessary as our algorithm would require drastic changes to the one in SPIN.

4. EXPERIMENTS

Our experiments evaluate the impact of the FTS algorithm on the runtime and the size of the state space. As SPL model checking is a novel problem, fPromela and SNIP cannot be compared directly to any existing tool. To conduct experiments, we implemented the enumerative algorithm with a script that transforms fPromela to Promela and calls SPIN and SNIP without the FTS algorithm (dubbed 'enum (snip)' in the statistics). 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. The results here are a representative subset of the collected data (available online[6]).

We considered a mine pump system [14] with a controller, a pump and a methane sensor. The system should prevent flooding of the mine; also the pump should not be switched on in case there is methane in the mine. The features cover the available components of the system. There are 11 features and 128 products; its FTS has 21,177 states, all products combined have 889,252. Another example is a subset of the CCSDS file delivery protocol (CFDP) [8], with 10 features and 30 products; its FTS has 3,809,320 states, all products combined have 4,460,038. The CFDP is a file delivery protocol for use in space missions. It is vast, and a mission usually only needs a subset of its functionality. The features correspond to the variations in the send and receive parts of the protocol: whether or not an entity operates in acknowledged mode, and in which type of acknowledged mode. These features were identified by Spacebel, an industrial parter with whom we collaborated on the development of a CFDP library SPL [2]. We also considered an elevator system [4], with two persons and four floors. The features cover variations in the behaviour of the elevator movement, its doors and buttons. There are 9 features and 256 products. The FTS of the elevator has 572,815 states, all products combined have 63,051,024.

For each case, we measured the time and number of states required for an exploration of the state space (assert and deadlock checking), and for checking four LTL properties (satisfiable and unsatisfiable). The results for the runtime are shown in the left column of Figure 3. The x-axis shows the property IDs in the supporting material. As can be seen, SNIP almost always beats the enumerative SNIP; in the case of the mine pump by a factor of at least three. Comparing the runtime to SPIN does not allow for any general conclusion as to the impact of the FTS algorithm. Still note that SNIP generally outperforms the enumerative algorithm with SPIN. SPIN spends most of the time compiling process anal-

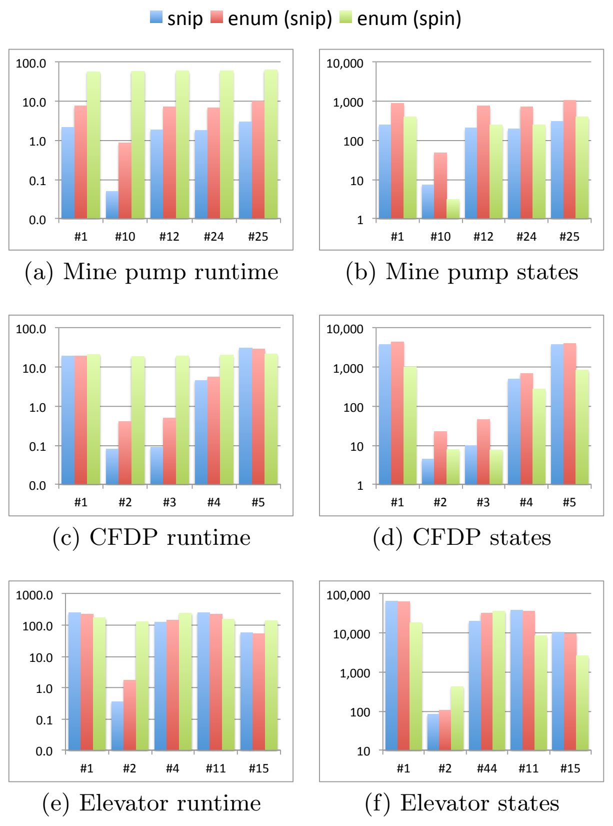
[2]www.lsv.ens-cachan.fr/~gastin/ltl2ba

[3]vlsi.colorado.edu/~fabio/CUDD

[4]www.minisat.se

[5]www.info.fundp.ac.be/~acs/tvl

[6]www.info.fundp.ac.be/~acs/snip/benchmarks



(a) Mine pump runtime (b) Mine pump states

(c) CFDP runtime (d) CFDP states

(e) Elevator runtime (f) Elevator states

Figure 3: Benchmark results. Runtime is in seconds, logarithmic scale. State space size is in thousands of states, logarithmic scale.

ysers. If this time is not counted, SPIN is generally more efficient. This is most likely due to its highly efficient code and optimisations developed for transition systems of single systems.

The results for the size of the state space are shown in the right column of Figure 3. An interesting observation is that the FTS algorithm, the only optimisation currently implemented in SNIP, often achieves a greater reduction of the state space than SPIN. The CFDP and the elevator system are both highly parallel. Partial order reductions of SPIN thus allow for greater reductions for some of their properties. These results show that the FTS algorithm is a viable approach for state space reduction, although there is room for improvement of the implementation. Furthermore, we believe that the state space reductions of SNIP and SPIN can be additive. Our ultimate goal is therefore to adapt existing optimisations for transition systems to the case of FTS. These experiments confirm that this is a promising and exciting area of future work.

5. CONCLUSION

We presented SNIP, the first complete toolset for SPL model checking. SNIP puts the theoretical results of [5] into practice and makes them available to engineers through an intuitive specification language for behaviour (viz. fPromela) and for feature models (viz. TVL). Most existing work on SPL verification [15, 16, 10] either lacks the specification language, the feature modelling language or has not been implemented at all (see comprehensive survey in [5, 4]).

6. REFERENCES

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

[2] Q. Boucher, A. Classen, P. Heymans, A. Bourdoux, and L. Demonceau. Tag and prune: A pragmatic approach to software product line implementation. In *ASE 2010*, pages 333–336. ACM, 2010.

[3] A. Classen, Q. Boucher, and P. Heymans. A text-based approach to feature modelling: Syntax and semantics of TVL. *Sci. Comput. Program.*, 76:1130–1143, 2011.

[4] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic model checking of software product lines. In *ICSE 33*, pages 321–330. ACM, 2011.

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

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

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

[8] Consultative Committee for Space Data Systems (CCSDS). *CCSDS File Delivery Protocol (CFDP): Blue Book, Issue 4*. Number CCSDS 727.0-B-4. NASA, 2007.

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

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

[11] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

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

[13] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *ICSE 30*, pages 311–320. ACM, 2008.

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

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

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

[17] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *ICSE 32, Proceedings*, pages 105–114, 2010.

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

APPENDIX (Demo)

The demo will revolve around two examples. First, a small introductory example (the one from Figure 1(a)) will be used to illustrate the main concepts of fPromela and TVL. This gentle introduction ensures that the listener will be able to follow the second example, which is the mine pump system (one of the examples used in the experiments in Section 4). The mine pump example was chosen because it is rather intuitive, has lots of interesting properties and offers a good trade-off between model size and response time.

Introductory example

The starting point of the demo is a brief introduction into software product lines (SPLs). A central concept in SPLs are features, and thus the introduction will focus on features and feature models, and how they affect modelling and development. In SNIP, feature models are specified with TVL [3], a textual modelling language. An example of a TVL feature model is the following.

```
root Example group [0..2] {
  Foo,
  Bar
}
```

This model specifies an SPL with three features: *Example*, *Foo*, and *Bar* and four valid products: *"Example"*, *"Example, Foo"*, *"Example, Bar"* and *"Example, Foo, Bar"*.

fPromela can be used to specify the behaviour of those products. An example of an fPromela model is the following.

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

The syntax of fPromela is the same as for Promela [11]. Except for conditional statements, most constructs are similar to C. This should make it easy to follow even for those who are not familiar with Promela. For those who are, we would like to point out 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.

Let us first examine lines 1-6. The features used in a file have to be declared as fields of the special type `features`. The reason for this is twofold: it serves as an interface that identifies the features used in the fPromela model and it ensures compatibility with Promela (a syntactically valid and well-typed fPromela file is also a syntactically valid and well-typed Promela file). The features can then be referenced by declaring any variable with this type (`f` in the example). Feature variables can only be read inside conditional statements; not written or printed.

The behaviour of the example system is specified at lines 8-18. A `proctype` denotes a process, and `active` means that it is started when the system starts. Several processes can be defined, and run concurrently. Processes can communicate through shared variables or channels. Variability in fPromela is specified by *guarding* statements. This choice of syntax is motivated by the pervasive use of #ifdefs when implementing variability in practice. The `i++` statement at line 2, for instance, is guarded with the expression `f.Foo || f.Bar` (line 11). This means that the `i++` statement is only part of products containing *Foo* or *Bar*. The other products (line 13) do nothing (line 14).

At line 15, the property that `i` equals one is specified using an assertion. Assertions are a very intuitive way to specify reachability properties. Alternatively, properties can be specified using LTL or directly as *never claims*.

Let us now illustrate SNIP and its user interface. As for most model checkers, SNIP's use consists in launching checks with certain parameters (property, execution bound, and so on). A very efficient interface for this is the command line; it remembers past commands and keeps a trace of inputs and outputs. SNIP is thus a command-line application. The list of its parameters is shown when launching SNIP.

To check the example given above, 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 output consists of two parts. First, SNIP reports the products for which the property is violated in the form of an expression over the feature variables (rather than a list of products, which could be huge). The advantage of returning such a *feature expression* is that it immediately identifies the features that are involved in that particular violation. Other model checkers would just return `false`. Second, SNIP gives a stack trace, that is, an execution of the fPromela model which proves the property violation. 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 0, of type `toto` is at line 11), and the values of the variables. To make traces shorter and easier to understand, variables are only printed if their value 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 traces: `-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 traces.

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. With the parameters used above, SNIP stops as soon as it finds a violation. At this point it is unclear whether the other products also satisfy or violate the property. The parameter `-exhaustive` causes SNIP to compute this as well.

```
$ ./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].
 - Products by which it is violated (as feature expression):
   (!Foo & !Bar)

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

In this mode, SNIP will print a violation upon finding it, and continues searching for violations in the other products. In the example, we disabled printing of traces using `-nt`, otherwise, SNIP will print a stack trace for each violation. When SNIP terminates, it will print a summary with all the products found to violate. 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.

Note the difference to using a classical model checker. In fact, SPIN could be used to check the model, but it would only check a single product (the one with no feature). SNIP, in contrast, checks all feature combinations (efficiently) and identifies those that contain a violation.

Mine pump example

The mine pump example is also distributed with SNIP. It consists of 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 model consists of a number of processes communicating over channels. The controller process is modelled after the CONIC code in [14]. It contains a large number of properties (42), including an explanation and satisfying products of each property. The demo will focus on one such property: *"There is never a situation in which the pump runs indefinitely even though there is methane."*; in LTL: `!<>[] (pumpOn && methane)`.

Checking this property with SNIP yields the following.

```
$ ./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)
```

Note that we did not specify the feature model explicitly. SNIP automatically looks for a file named `minepump.tvl`. 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 stack traces reveals a problem with the property. Basically, the controller has a central loop, in which it can receive three types of messages: user commands (start and stop), methane alarm messages, and water level readings. The stack traces 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))`.

```
$ ./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. A step which we did not show here is to discharge the assumption. This is done by checking its negation. In this case, the assumption is discharged by all products.

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. It is therefore sensible to check it only against products that have the feature. This can be accomplished in SNIP using the `-filter` parameter. It can be used to restrict the verification to a subset of all products specified as a feature expression (in TVL syntax). Limiting the previous check to products with *MethaneAlarm* yields the following.

```
$ ./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. This concludes the demo.

Availability and maturity

SNIP is available at www.info.fundp.ac.be/~acs/snip. It has been tested under Mac OS X and Ubuntu. SNIP requires a UNIX-like environment, the *gcc* compiler, the *cpp* preprocessor and Java (for the TVL library). The tool is mature and well-documented. SNIP's development continues and new versions are posted with a changelog on the website.