ProVeLines: A product-line of Verifiers for Software Product Lines

Conference Paper · August 2013

DOI: 10.1145/2499777.2499781

CITATIONS
61

READS
290

4 authors, including:

Maxime Cordy
University of Luxembourg
105 PUBLICATIONS **1,106** CITATIONS
SEE PROFILE

Pierre Yves Schobbens
University of Namur
228 PUBLICATIONS **4,492** CITATIONS
SEE PROFILE

Patrick Heymans
University of Namur
208 PUBLICATIONS **6,465** CITATIONS
SEE PROFILE

ProVeLines

A Product Line of Verifiers for Software Product Lines

Maxime Cordy*
PReCISE Research Center
University of Namur, Belgium
mcr@info.fundp.ac.be

Andreas Classen
PReCISE Research Center
University of Namur, Belgium
acs@info.fundp.ac.be

Patrick Heymans
PReCISE Research Center,
University of Namur, Belgium.
phe@info.fundp.ac.be

Pierre-Yves Schobbens
PReCISE Research Center,
University of Namur, Belgium.
pys@info.fundp.ac.be

Axel Legay
INRIA Rennes, France
axel.legay@inria.fr

ABSTRACT

Software Product Lines (SPLs) are families of similar software products built from a common set of features. As the number of products of an SPL is potentially exponential in the number of its features, the model checking problem is harder than for single software. A practical way to face this exponential blow-up is to reuse common behaviour between products. We previously introduced Featured Transition Systems (FTS), a mathematical model that serves as a basis for efficient SPL model checking techniques. In this paper, we present ProVeLines, a product line of verifiers for SPLs that incorporates the results of over three years of research on formal verification of SPLs. Being itself a product line, our tool is flexible and extensible, and offers a wide range of solutions for SPL modelling and verification.

Categories and Subject Descriptors

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

General Terms

Theory, Verification

Keywords

Software Product Lines, Model Checking, Tool, Features

1. INTRODUCTION

Variability has become ubiquitous in today's systems, be it in the form of configuration options or extensible architectures. By mastering variability, developers can adapt their system to new or changing requirements without having to develop entirely new applications. Software Product Line engineering (SPLE) is a software development paradigm that focuses on the development of a *family* of similar software products (also called "variants"). In SPLs, variability is commonly represented in terms of *features*, *i.e.*, units of difference between products that appear natural to stakeholders and technicians alike. The products of an SPL are therefore defined by the set of their features. Interdependencies between features (*e.g.*, exclusion) are commonly captured in a *Feature Model* (FM) [19], which specifies which combinations of features are valid.

SPLE promotes systematic reuse through all the development stages, which leads to significant economies of scale and reduced time to market. These benefits, however, come at the cost of a new source of complexity: variability has to be managed throughout the software lifecycle. This poses a new range of theoretical *and* practical problems. In particular, quality assurance for SPLs is harder than for single systems because engineers have to provide solid evidence that *all* the products they build satisfy intended properties.

Model checking is an established automated technique for verifying system behaviour. To verify a product line, one can apply single-system model checking to *each* software variant. However, the number of products in an SPL is potentially exponential in the number of features. This *enumerative* approach is thus impractical in industrial SPLs, where hundreds of variants are built. Therefore, there is a need for novel techniques that can exploit the commonalities between the products to reduce the verification effort.

To combat this exponential blow-up, we proposed Featured Transition Systems (FTS) [11], a variability-aware extension of transition systems that represents the behaviour of all the products in a single compact model. We designed efficient algorithms to verify FTS against properties expressed in Linear Temporal Logic (LTL) [9]. Those exploit the variability information captured in the FTS to verify common behaviour across several products only once, which significantly reduces verification time. We implemented our algorithms in SNIP, an SPL model checker we developed from scratch [8].

Driven by recent advances in single-system verification as well as by practical needs of SPL engineers, we recently extended FTS and its associated theory in various directions. We designed algorithms to check *equivalence* between SPL

*FNRS Research Fellow

SPLC 2013 workshops, Aug 26-30 2013, Tokyo, Japan

behavioural models [12]. We extended our model checking algorithms to verify *real-time* SPLs [13]. Most recently, we introduced an approach to cope with *multi-features* – features that may appear several times in a product (e.g., processing units which number is variable from one product to another and which can be configured independently) – and *numeric features* (*e.g.*, maximum number of users) [14].

A solution to implement all these results is to extend SNIP. Unfortunately, there exist no unique data structure to represent and manipulate efficiently all the required constructs. Instead, these constructs have to be encoded into separate data structures which manipulation increases the overall verification time. It thus appears more efficient to design, for each subset of these constructs, a new variant of the tool that manipulates only the required data structures. These variants share many commonalities in source code, architecture, specification language, and algorithms. Moreover, new variants will be developed to implement future FTS-based approaches. The need for managing all the existing and future variants motivated us to re-engineer SNIP as a product line named ProVeLines and of which SNIP is but one of the products. The tool variants are built from the same code base by pruning code and setting compilation options. ProVeLines is the first SPL model checking toolset to provide all the aforementioned functionalities. It is open-source and freely available on our website.[1]

2. THEORETICAL FOUNDATIONS

Model checking is an automated technique for verifying system behaviour. The two ingredients of this verification method are (1) an executable behavioural model of the system, typically a *transition system*, and (2) a property expressed in temporal logic. A model-checking procedure consists in systematically exploring every execution path of the model and checking whether each of them satisfies the property [3]. If that is not the case then the model checker returns a example of execution path that violates the property.

The model-checking problem for SPLs is harder than for single systems, as one has to identify which variants do not satisfy an intended property. An immediate solution is to model *each* product as a distinct transition system, and to verify each of them with a single-system model checker. This *enumerative* approach checks a given execution path as many times as the number of products that can exhibit it. This is clearly suboptimal and goes against the principles of SPLE, which promotes systematic reuse.

As an alternative, we proposed an approach based on *Featured Transition Systems* (FTS). FTS are an extension of transition systems where transitions are labelled with *feature expressions*, *i.e.* propositional formulae over the features. A feature expression encodes the set of products able to execute the associated transition. The transition system modelling a particular product is obtained by removing the transitions that this product cannot execute. An FTS is thus ***a compact behavioural model of a set of products***.

An excerpt of FTS is shown in Figure 1. It depicts the behaviour of a motor. The motor is initially in state `running` and remains therein as long as there is no danger. When danger occurs, the motor should stop. However, the system cannot detect danger without feature *Alarm*. It thus can stop iff this feature is enabled (see transition `[`*Alarm*`]danger` from `running` to `stopped`). Otherwise, it remains in state `running` (see self-transition `[`¬*Alarm*`]danger` on `running`).

[1] `http://info.fundp.ac.be/fts/provelines`

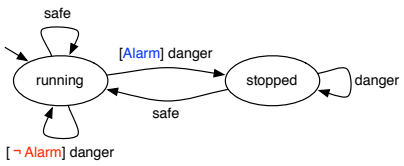

Figure 1: An example of FTS

In past work [11, 10, 12, 13, 9], we proposed efficient algorithms to verify FTS. While exploring an execution path, our algorithms keep track of the set of products able to execute it. These are those that satisfy the conjunction of the feature expressions labelling the executed transitions. By doing so, the algorithms ***check an execution path only once, regardless of how many products can execute it***. This set of products can be empty, for instance when the feature expression of a transition is the negation of the feature expression of another transition. In our example, this happens if `[`¬*Alarm*`]danger` and `[`*Alarm*`]danger` are triggered consecutively. To avoid exploring such paths, our algorithms check the satisfiability of the feature expression that characterizes the execution path. Evaluations carried out in our previous papers tend to show that our FTS-based algorithms outperform the enumerative approaches.

Although all our algorithms are based on the above principles, they still differ in various ways. We first designed an algorithm to verify an FTS against properties expressed in Linear Temporal Logic (LTL) [3]. We introduce the basic procedure in [11] and we present optimizations in [9]. Rather than properties verification, we focus in [12] on checking behavioural inclusion and equivalence between products modelled in two FTS. This approach is based on *F-simulation*, a variability-aware extension of simulation relation [3]. F-simulation allows one to compare an abstract (smaller) FTS with respect to the original FTS. Again, this verification problem is more complicated than its single-system counterpart since we have to identify which products have their behaviour preserved by abstraction. In [13], we tackle the model-checking problem for real-time SPLs. The main results of this paper is the definition of featured timed automata – a formalism that enriches FTS with real-time information – and an algorithm to check such models against real-time properties. Recently, we extended our formalisms to support multi-features and numeric features [14]. Although this extension does not change the core verification procedure, it increases the complexity of feature expressions, and thereby the verification time.

3. OVERVIEW OF PROVELINES

ProVeLines is the realization of all our model-checking approaches into a unified implementation. To better cope with variability and facilitate extensibility, we re-engineered the architecture of SNIP. Figure 2 shows the new architecture of ProVeLines. Any ProVeLines variant requires at least two artefacts from the user: an FM and an fPromela model. For the former, we use TVL [7, 14], one of the latest incarnations of FMs, due to some of its advantages: high expressiveness, formal semantics and tool support. The FM

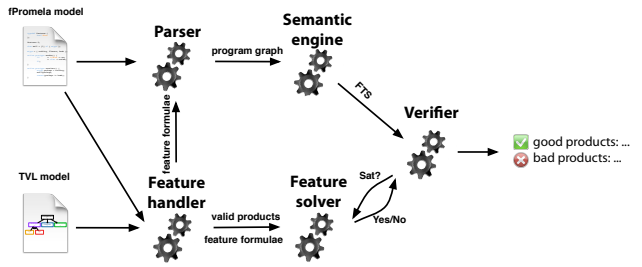

Figure 2: Architecture of ProVeLines

allows us to ***limit the verification to the valid products only***; the checker will ignore execution paths that no valid products can execute. fPromela is a feature-oriented extension of Promela [18], which we defined as a ***high-level language on top of FTS***. An fPromela model thus describes the behaviour of all the products defined by the FM [8, 14]. An overview of fPromela constructs is provided in Appendix.

The fPromela model is parsed by the `Parser` component, which builds a program graph [3] where transitions are annotated with feature expressions. Feature expressions are built by the `Feature handler`, *i.e.* a set of wrappers to data structures to represent propositional formulae (*e.g.* binary decision diagrams [5], abstract syntax tree). The parser transmits the program graph to the `Semantic engine`, which constructs on the fly the FTS corresponding to the program graph. The semantic engine gives the FTS as input to `Verifier` which triggers the verification. Meanwhile, the feature handler communicates with a TVL tool to extract a formula encoding the set of valid products. The formula is sent to `Feature solver`. The role of the latter is to prevent the exploration of execution paths that are available to none of the valid products. Once the verification is complete, the verifier summarizes the results and displays ***a concise formula representing the products that contain errors***.

Developing ProVeLines as an SPL allows us to ***tailor our tool to the specific needs of the user***. Indeed, many of its features augment the capabilities of the tool but increase the verification time. In a specific variant, the unneeded features should thus be removed. We organize the features of ProVeLines in an FM shown in Figure 3. Variability within ProVeLines originates from four factors: the `System`, the `Property`, the supported types of `Features`, and the `Data Structure`. Altogether, ProVeLines consists of 16 unique products, including SNIP.

ProVeLines can check purely `Discrete` and `Real-Time` models. To support the latter, modifications are required in the parser (to handle timed statements in fPromela, see Appendix), the semantic engine (to encode timed statement into real-time data structures), and the verifier (to make the verification aware of real-time, that is, to consider featured timed automata instead of FTS). ProVeLines can perform three types of computations: `Reachability`, F-`Simulation`, and `LTL` checking. Optionally, one may check `Stutter` F-simulation instead of standard F-simulation. When real-time models are considered, ProVeLines is limited to reachability. Accordingly, feature Real-time cannot coexist with features LTL and Simulation. All the verification algorithms are implemented as part of the verifier.

In addition to Boolean features, ProVeLines supports `Multi-Features` and `Numeric Features`. As features have to be declared in the fPromela model, the support for non-Boolean features requires modifications to the parser. The feature handler and solver have to be modified as well since the data structures that encode feature expressions depend on the types of considered features. Variability related to system types, algorithms, and features is implemented in the form of preprocessor directives, *e.g.*, `#ifdef`.

ProVeLines provides three data structures to represent feature expressions and check their satisfiability. For Boolean features, ProVeLines can use either Binary Decision Diagrams (`BDD`) [5] (implemented in the CuDD[2] library) or formulae in Conjunctive Normal Form (`CNF`) checked by MiniSat [16]. At this time, our case studies tend to show that BDDs are more efficient than CNFs but further empirical studies are needed to confirm this observation. To handle numeric features, we developed a wrapper to Microsoft's `Z3` [15]. In Z3, formulae are encoded in native data structures called Abstract Syntax Trees (`AST`). ProVeLines uses them to encode feature expressions and calls Z3's algorithms to check their satisfiability. The choice of a data structure is applied via compilation options. To encode real-time, ProVeLines relies on UPPAAL's implementation of Difference Bound Matrices (`DBM`) [4] combined with feature information. Observe that SNIP is the variant with features Discrete, LTL, Reachability, and BDD.

4. RELATED WORK

There exist tools similar to ProVeLines. In our earlier work, we developed an extension of NuSMV [6] that can verify SPLs modelled in the fSMV language [20]. This tool uses the fully symbolic FTS algorithm [10], whereas ProVeLines implement semi-symbolic FTS algorithms where states are represented explicitly rather than symbolically.

Other tools for SPL verification were developed in the past years. Asirelli *et al.* [2] enriched modal transition systems (i.e., transition systems with mandatory and optional transitions) with a logic able to link behaviour to features. This logic allows them to derive the transition system corresponding to a particular product. They implemented a model checker that applies an enumerative approach to verify each product [23].

Post and Sinz [21] proposed another approach based on a technique called lifting. It consists in incorporating variability information about the verifiable model. A similar approach is followed by SPLVerifier [1], in which features are modelled as separate and composable units. Both approaches use a single-system model checker to detect violations. Thereby, the verification stops once a violating product is found. It thus cannot compute the products that satisfy the property.

5. CONCLUSION

ProVeLines is a product line of model checkers for SPLs. We designed its architecture in a way that makes it extensible. It will serve as a basis for the implementation of future FTS-based verification methods. In particular, we will broaden the portfolio of our checking algorithms, enabling the verification of stochastic and hybrid SPLs. The major challenge in these new approaches is to define efficient data

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

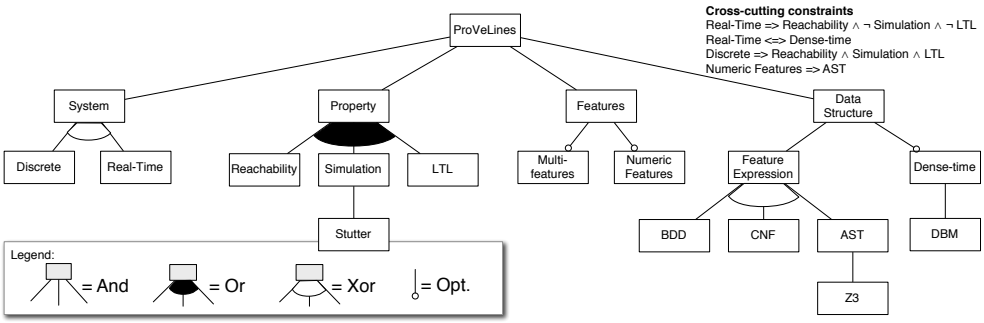

Figure 3: Features of ProVeLines

structures to represent and manipulate quantitative data. We will also explore alternatives to our search-based algorithms, such as compositional reasoning and distributed algorithms. To increase the efficiency of our algorithms, we will not only lift classic optimizations (*e.g.*, partial-order reduction) to FTS, but also design SPL-specific optimizations. Finally, to make our approach accessible to engineers, we will link ProVeLines to other high-level languages. Among those, we will study graphical representations combined with variability [22, 17].

6. REFERENCES

[1] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Feature-interaction detection using feature-aware verification. In *ASE'11*, pages 372–375. IEEE, 2011.

[2] P. Asirelli, M. H. ter Beek, A. Fantechi, and S. Gnesi. Formal description of variability in product families. In *SPLC'11*, pages 130–139. Springer-Verlag, 2011.

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

[4] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL in 1995. In *TACAS'96*, pages 431–434. Springer-Verlag, 1996.

[5] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, Sept. 1992.

[6] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *CAV '02*, volume 2404 of *LNCS*. Springer, July 2002.

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

[8] A. Classen, M. Cordy, P. Heymans, A. Legay, and P.-Y. Schobbens. Model checking software product lines with SNIP. *STTT*, 14(5):589–612, 2012.

[9] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. cois Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. *Transactions on Software Engineering (in press)*, 2013.

[10] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic model checking of software product lines. In *ICSE'11*, pages 321–330. ACM, 2011.

[11] 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'10*, pages 335–344. ACM, 2010.

[12] M. Cordy, A. Classen, G. Perrouin, P. Heymans, P.-Y. Schobbens, and A. Legay. Simulation-based abstractions for software product-line model checking. In *ICSE'12*, pages 672–682. IEEE, 2012.

[13] M. Cordy, P. Heymans, P.-Y. Schobbens, and A. Legay. Behavioural modelling and verification of real-time software product lines. In *SPLC'12*. ACM, 2012.

[14] M. Cordy, P.-Y. Schobbens, P. Heymans, and A. Legay. Beyond boolean product-line model checking: Dealing with feature attributes and multi-features. In *ICSE'13*, pages 472–481. IEEE, 2013.

[15] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS'08*, pages 337–340. Springer-Verlag, 2008.

[16] N. Eén and N. Sörensson. An extensible sat-solver. In *SAT'03*, pages 502–518. Springer, 2003.

[17] J. Greenyer, A. M. Sharifloo, M. Cordy, and P. Heymans. Efficient consistency checking of scenario-based product line specifications. In *RE '12*, pages 161–170, 2012.

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

[19] 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, 1990.

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

[21] H. Post and C. Sinz. Configuration lifting: Verification meets software configuration. In *ASE'08*, pages 347–350. IEEE CS, 2008.

[22] P. Shaker, J. M. Atlee, and S. Wang. A feature-oriented requirements modelling language. In *RE '12*, pages 151–160, 2012.

[23] M. H. ter Beek, F. Mazzanti, and A. Sulova. Vmc: A tool for product variability analysis. In *FM '12*, pages 450–454, 2012.

APPENDIX: Roadmap of the Demonstration

The demonstration begins with a quick reminder of the theoretical foundations on which ProVeLines is built. The content of this presentation is similar to Section 2. Next, we propose a gentle introduction to fPromela, its various constructs, and the functionalities of ProVeLines. Each particular construct is illustrated through small examples and is linked to ProVeLines's features that support it.

Introductory Examples

A first difference between fPromela and Promela is that the former distinguishes features from normal variables. Features are declared in a specific user-defined data structure called `features`. In its most basic form, this data structure contains only Boolean fields:

```
// feature declaration                                    1
typedef features {                                        2
   bool Foo;                                              3
   bool Bar                                               4
};                                                        5
features f;                                               6
```

Then the user can use these variables to specify that statement blocs are specific to some products. For example, the following excerpt specifies that the process `toto` will increment variable `i` iff features `Foo` or `Bar` are enabled:

```
// process declaration                                    1
active proctype toto() {                                  2
  int i = 0;                                              3
  // guarded increment stmt                               4
  gd :: f.Foo || f.Bar -> i++;                            5
     :: else           -> skip;                           6
  dg;                                                     7
  // test assertion                                       8
  assert(i == 1);                                         9
}                                                         10
```

ProVeLines can detect feature variables and automatically translate them into feature expressions. This is the semantic difference between fPromela and Promela. The latter considers feature variables as normal variables, which impedes the use of the efficient algorithms offered by FTS.

Once the user has built a TVL and an fPromela model, she can trigger the verification of a property via ProVeLines command-line interface (check our website for a complete description of the command-line options). During the verification, ProVeLines will apply our efficient algorithms to identify all the products that violate the property. These products will appear in the form of a concise formula that represents the combinations of features responsible for the violation. *E. g.*, ProVeLines can detect that the above assertion is violated if none of the two features are enabled:

```
$ ./provelines -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 addition to property checking, we can also verify whether an abstraction of the above model preserves its behaviour. For example, let us consider a variant of this model that makes abstraction from feature `Bar`:

```
active proctype toto() {                                  1
  int i = 0;                                              2
  gd :: f.Foo -> i++;                                     3
     :: else  -> skip;                                    4
  dg;                                                     5
  assert(i == 1);                                         6
}                                                         7
```

Then, we check for which products the abstract model simulates (that is, includes the behaviour) of the original model:

```
$ ./provelines -props p.prop -sim Orig.pml Abst.pml
Checking F-simulation..
Simulation holds for the following products:
  (!Foo & !Bar) | (Foo)
```

As before, ProVeLines reports a formula describing the result in terms of features. In our example, it turns out that simulation does not hold iff feature `Bar` is present without feature `Foo`. Indeed, in this case the original model increments variable `i` whereas the abstract model does not. We can check the inverse question, that is, whether the original model simulates the abstract model:

```
$ ./provelines -props p.prop -sim Abst.pml Orig.pml
Checking F-simulation..
Simulation holds for the following products:
  (!Foo & !Bar) | (Foo)
```

We conclude that the two models are simulation-equivalent for products that have either feature `Foo` or none of the two features.

Most recently, we incorporated timed statements in fPromela to specify that the system needs time to execute the next action. Let us consider the following example:

```
typedef features {                                        1
   bool FastStart;                                        2
   bool FastStop                                          3
};                                                        4
features f;                                               5
// clock declaration                                      6
clock c;                                                  7
bool on = false;                                          8
active proctype toto() {                                  9
  off: on = false;                                        10
  // timed statements                                     11
       gd :: f.FastStart  ->                              12
             while (c<7) wait;                            13
             when (c>4) reset(c) goto on;                 14
          :: else          ->                            15
             while (c<10)        wait;                    16
             when (c>7) reset(c) goto on;                 17
       dg;                                                18
                                                          19
  on: ...                                                 20
}                                                         21
```

We use real-time clocks to model that time elapses (see Line 7). Statement *wait* specifies that the process can delay its next action as long as the value of a given clock remains in a certain interval. Statement `when` specifies that the process can trigger the next transition only when the value of a given clock lies in a certain interval. For example, Lines 14–15 specify that when feature `FastStart` is enabled the system needs between 4 and 7 time units to move from state `off` to state `on`. When this feature is disabled, the delay amounts to between 7 and 10 time units. We can check timed fPromela models against timed reachability properties such as "for which products can the system be turned on in less than 5 time units". As expected, ProVeLines finds out that feature `FastStart` is required to satisfy the property:

```
$ ./provelines -check -tctl 'EF (<5) on' -fm
   features.tvl model.pml
```

```
Checking timed CTL property EF (<5) on..
Property violated
 - Products by which it is violated (as feature
   expression):
   (!FastStart)
```

Another recent extension of fPromela is the support for multi-features and numeric features. The former are declared as array fields in the `features` data structure. For instance, the following excerpt defines that *Foo* is a multi-feature with at most two instances:

```
typedef features {                                  1
   bool Foo[2];                                     2
   bool Bar                                         3
};                                                  4
features f;                                         5
```

Each instance of a multi-features can be associated to a specific instance of a process. In the following, each instance of process `toto` is associated to a distinct instance of *Foo*:

```
active[card(foo)] proctype toto() {                 1
   ...                                              2
}                                                   3
```

The behaviour of each process can be different, as it depends on the associated instance of *Foo* and its subfeatures. To support numeric features, fPromela allows non-Boolean fields to occur inside the `features` data structure:

```
typedef features {                                  1
   someType Foo                                     2
};                                                  3
typedef someType {                                  4
   bool is_in;                                      5
   int attribute                                    6
};                                                  7
features f;                                         8
```

More precisely, a numeric feature is declared as a data structure which first field is Boolean and named `is_in`. Its value encodes the presence or absence of the feature. The other fields are numeric and contain the quantitative data of the feature.

After the presentation of the main constructs of fPromela, we illustrate the use of ProVeLines on a practical example.

Minepump Case Study

We consider an SPL of minepump systems [11, 9]. Such a system 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 is composed of several communicating processes. The demo will focus on the following 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 ProVeLines yields the following.

```
$ ./provelines -check -exhaustive -nt
  -ltl '!<>[] (pumpOn && methane)' minepump.pml

Checking LTL property !<>[] (pumpOn && methane).
Property violated [explored 469 states, re-explored 0]
 - Products by which it is violated (as feature
 expression):
   (Start & Stop & MethaneQuery & MethaneAlarm & Low
   & High)

[...]

Exhaustive search finished  [explored 13199 states,
re-explored 110745].
 - 16 problems were found covering the following
 products (others are ok):
(Start & High)
```

Note that we did not specify the feature model explicitly; in this case, ProVeLines automatically looks for a file named `minepump.tvl`. ProVeLines 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 all the products. 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, 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 nondeterministic, 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 as the LTL formula: `(([]<> readCommand) && ([]<> readAlarm) && ([]<> readLevel))`.

```
$ ./provelines -check -exhaustive -nt -ltl
  '([]<> read..) -> (!<>[] pump..)' minepump.pml

Checking LTL property ([]<> read..) -> (!<>[] pump..).
Property violated [explored 20674 states, re-explored
92326]
 - Products by which it is violated (as feature
 expression):
   (Start & Stop & MethaneQuery & !MethaneAlarm & Low
   & High)

[...]

Exhaustive search finished  [explored 26380 states,
re-explored 197637].
 - 8 problems were found covering the following
 products (others are ok):
(Start & !MethaneAlarm & High)
```

According to this result, feature *MethaneAlarm* is required to satisfy the property. This corresponds to what we expected, as feature *MethaneAlarm* alerts the controller of methane, leading it to shut off the pump.

Normally, the example property is not expected to hold for products that do not have feature *MethaneAlarm*. It corresponds to a requirement implemented by the feature. We should therefore check it only against products that have the feature. This can be accomplished in ProVeLines using the `-filter` parameter. This parameter restricts 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.

```
$ ./provelines -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 21201 states, re-explored
188589].
```

The property is indeed satisfied by all relevant products. This concludes the demonstration.