An extensible platform for product-line behavioural analysis

Cordy, Maxime; Willemart, Marco; Dawagne, Bruno; Heymans, Patrick; Schobbens, Pierre-Yves

Published in:
ACM International Conference Proceeding Series

DOI:
10.1145/2647908.2655973

Publication date:
2014

Document Version
Publisher's PDF, also known as Version of record

An Extensible Platform for Product-Line Behavioural Analysis

Maxime Cordy*
PReCISE Research Center
University of Namur, Belgium
maxime.cordy@unamur.be

Marco Willemart
PReCISE Research Center
University of Namur, Belgium
marco.willemart@unamur.be

Bruno Dawagne
PReCISE Research Center
University of Namur, Belgium
bdawagne@student.unamur.be

Patrick Heymans
PReCISE Research Center
University of Namur, Belgium
patrick.heymans@unamur.be

Pierre-Yves Schobbens
PReCISE Research Center
University of Namur, Belgium
pierre-yves.schobbens@unamur.be

ABSTRACT

Software Product-Line (SPL) model checking has reached an adequate level of efficiency and expressiveness to be applied on real-world cases. Yet a major challenge remains: model checkers should consist of black-box tools that do not require in-depth expertise to be used. In particular, it is essential to provide engineers with easy-to-learn languages to model both the behaviour of their SPL and the properties to check. In this paper, we propose a framework to build customized product-line verifiers modularly. Our extensible architecture allows one to plug new modelling languages or verifications algorithms without modifying other parts of it. It also provides means of representing and reasoning on variability that can facilitate the development of other SPL quality assurance techniques. We illustrate the benefits of our approach by detailing how we created a new domain-specific SPL modelling language and linked it to our tool.

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

*FNRS Research Fellow

SPLC'14, September 15 - 19 2014, Florence, Italy

http://dx.doi.org/10.1145/2647908.2655973.

1. INTRODUCTION

Software Product-Line (SPL) engineering is a software development paradigm that aims at the efficient development of similar products. It considers these products as members of a family (also called *variants*) built from a common set of assets. Systematically reusing common assets allows economy of scale and a reduced time-to-market [19]. These benefits, however, come at the cost of an additional source of complexity: the difference between the products, i.e. the *variability*, has to be managed throughout all the development phases. In particular, traditional quality assurance techniques cannot cope with variability; they can be performed on individual products only. This is clearly inefficient, if not unfeasible, in case of real-world SPLs that consist of thousands of products or more.

Among the most popular quality assurance techniques, one finds model checking [13, 6], an established method for exhaustively verifying a behavioural model of a system against a temporal property. Transposed to the context of SPLs, the model checking problem requires identifying all the products that violate the property [18]. Applying single-system model checking on all variants separately would be suboptimal, for these may exhibit identical behaviour. As an alternative, variability-aware model-checking techniques were proposed [18, 17, 3, 5]. Their strength lies in their capability to represent and detect commonalities between products, thereby avoiding the verification of the same behaviour more than once. Early experiments suggest that SPL-specific heuristics provide substantial performance gains over an enumerative application of their single-system counterpart [15, 16, 4].

One of the most promising approach for SPL model checking is based on *Featured Transition Systems* (FTS) [18, 17]. FTS are transition systems whose transitions are annotated with constraints specifying the products that can execute these transitions. We developed efficient algorithms that exploit this information to facilitate the verification of properties [16]. These techniques were extended in many directions, including the support for real-time [21] and stochastic [22] SPL behaviour, as well as non-Boolean variability [23]. We implemented them into several tools [17, 15, 24]. ProVeLines is the latest incarnation of the SPL model checker we built [24]. It implements all the developments on FTS-based SPL verification, and is actually designed as a

product line itself. Thanks to the efficiency of its algorithms and the expressiveness of its formalisms, ProVeLines is now ready to be confronted to real-world SPLs.

Given that "verification using model-based techniques is only as good as the model of the system" [6], applying model checking in industry requires providing engineers with easy-to-learn languages to model both the system and the properties in a correct way. FTS and temporal logics like linear temporal logic [38] are fundamental formalisms, which are not meant to be used directly by engineers. Most of existing SPL verifiers provide user-friendly specification languages that hide the complexity of the underlying theory. In ProVeLines, these languages are dialects of Promela, the SPIN model checker's input language [32], which extend the original language with constructs to express variable behaviour. Although they constitute a first improvement in usability, Promela's extensions are not convenient for engineers not acquainted to such specification languages. On the other hand, several behaviour modelling languages commonly used in industry were extended to support variability [40, 28, 29] but have not been linked to a verification formalism like FTS. As alternatives, domain specific languages can be designed to model SPL behaviour and tailored to engineers' specific needs and preferences.

When we attempted to equip ProVeLines with additional input languages, we faced several obstacles. First, as most model checkers do for efficiency reasons, ProVeLines' verification algorithms are tightly coupled with its current input language, *viz.* Promela. Because of that, the addition of a completely different language requires to implement all the verification algorithms anew. Moreover, its variability is implemented using #ifdef statements into the code, a classic way of managing variability in C or C++; we were rapidly aware that the implementation of new features was becoming increasingly complex. More generally, our experience revealed that the growing number of variants (it currently has 166) makes evolution and maintenance cumbersome. Finally, more extensibility is needed for the integration with third-party applications to be feasible.

In order to overcome the aforementioned challenges and limitations, we propose a redesigned, extensibility-focused version of ProVeLines. Its modularized architecture facilitates the integration of new input languages, the customization or replacement of model-checking algorithms, and the reuse of variability-encoding modules. Our approach relies on the use of the object-oriented paradigm, as well as a *loosely coupled* architecture, made up of modules exposing data types and operations through narrow interfaces. Not only this new version is a stand-alone model checker whose each part can be customized at will and modularly, but it is also a software development kit for SPL verifiers. Overall, we claim that our tool has become a framework for the construction of efficient SPL verification tools.

The remainder of this paper is structured as follows. Section 2 briefly introduces verification techniques of SPLs. In Section 3, we describe the variability of ProVeLines and how we implemented it. Section 4 presents our new modular and extensible architecture. Section 5 explains the mechanisms we used to compose all the modules together. Section 6 details the step-by-step process for extending ProVeLines with a new input language. Finally, Section 7 reviews related work.

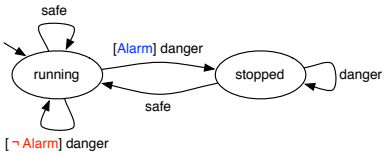

Figure 1: An example of FTS

2. BACKGROUND

Model checking is an automated technique for verifying system behaviour. Basically, a model checker systematically explores the execution paths of a behavioural model – typically a *transition system* – in search for violations of a property expressed in temporal logic. If it actually finds one, the model checker returns an example of execution path that yields the violation. The variability in SPLs creates a new dimension of complexity in the model-checking problem, 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 since it is sufficient to check a given behaviour only once.

In order to exploit the commonality between SPL products during verification, we proposed an alternative 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 an 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`).

Together with the above formalism, we designed efficient algorithms able to associate every execution path in an FTS with the exact set of products able to produce it. As opposed to the enumerative approach, our algorithms can avoid exploring an execution path as many times as there are products able to execute it. They also avoid verifying execution paths that no variant can execute. This happens when the feature expression of two successive transitions are not compatible. Two examples of such transitions are [¬*Alarm*]`danger` and [*Alarm*]`danger`. Incompatibility of two transitions is detected by checking the satisfiability of the conjunction of their feature expression. Previous evaluations tend to show that our FTS-based algorithms outperform the enumerative approaches [15, 16, 23].

ProVeLines [24] is a model checker we developed that implements a range of FTS-based techniques. Its input and output are shown in Figure 2. It takes as input a feature

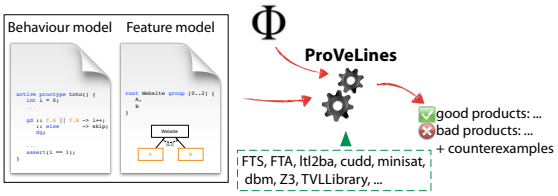

Figure 2: ProVeLines' input and output

model defining the valid products of the SPL, a behavioual model of these products, and a property to verify. Then, using its embedded formalisms and algorithms together with external libraries, ProVeLines determines which valid products satisfy the property and which do not. For each of the latter, it also returns a counterexample of execution that shows a violation executable by this product. In addition to the initial FTS model-checking algorithms [18], it is also able to check real-time [21] and stochastic [22] variants of FTS. It also supports non-Boolean variability [23] and alternative verifications such as (bi)simulation [20]. All these methods share many commonalities in source code, architecture, input, and algorithms. This is the reason why ProVeLines was engineered as a product line. 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 this many functionalities.

3. PROVELINES' VARIABILITY

Being itself a product line, ProVeLines includes several features that can be activated or deactivated. These features, along with the hierarchies and dependencies that exist between them, are specified in the feature model shown in Figure 3. This feature model is similar to the one presented in our previous work [24]; still, there exist notable differences that are definitely worth highlighting. In particular, the feature model has been reworked as to reflect the separation of concerns we emphasize in Section 4. Also, ProVeLines can now model and verify stochastic SPLs, which was not the case previously [24].

The variability of ProVeLines originates from five main factors, which uniquely correspond to the APIs described previously: the type of system to verify; the input language used to model its behaviour; the properties to check and the associated algorithms; the representation and expressiveness of feature models; the data structures to encode variability.

ProVeLines allows one to verify three types of SPL: discrete (i.e. whose state evolves at discrete time steps and is non-random), real-time, and stochastic. To compactly represent the behaviour of the SPL products, one can either use an *FSTM* or variability-aware variants of *Promela*. FSTM stands for *Featured State Machines*. It is a domain-specific language based on Harel's statecharts [31] we developed specifically for one of our industrial partners. Promela is the input language of the SPIN model checker [32]. Its syntax is close to that of imperative programming languages such as C. We extended it with additional constructs to guard statements with feature expressions, which allows one to restrict the set of products able to execute these statements. Depending of the chosen variant, it also provides specific statements to represent real time and probabilities. SPL behaviour can also be modelled in a feature-aware extension of Stateflow [33], a language to specify the stateful behaviour of reactive systems which is part of the Simulink environment.

The specified system can be checked against different types of properties. A first type is the properties whose verification is based on reachability computation. An example of such property is deadlock freedom. One can also check real-time properties specified in a fragment of the timed computational tree logic [2], whose computation comes down to reachability. Similarly, stochastic properties are expressed in a fragment of the probabilistic computation tree logic [30], which, again, is limited to reachability. For discrete systems, we allow one to verify LTL formulae. We also implemented algorithms to check behavioural inclusion between two FTS based on simulation relations, a prerequisite for reliable abstraction [20].

Another point of variability is the expressiveness needed to represent an SPL's variability, as well as the concrete syntax that represents a feature model. In addition to standard, Boolean features, ProVeLines supports two additional types that are intensively used in practice [23] : *multi-features* (i.e., features that may appear several times in a given product) and *numeric features*. The former entails changes only in feature model semantics. The latter, however, also raises the need for non-Boolean feature expressions.

Since our tool is meant to check variability-intensive systems, an important part of it is the way variability (viz. feature expressions) are encoded. Feature expressions can be internally represented and checked for satisfiability using Binary Decision Diagrams (BDDs), SAT solvers or SMT solvers; the latter is needed in case of numeric features. The current implementation of BDDs makes use of the CU Decision Diagram (CUDD) library. The back-end solvers we use are Minisat for SAT, and Z3 for SMT. Since the former does not provide data structures to represent boolean formulas, the features expressions are in this case encoded in an ad-hoc Abstract Syntax Tree (AST) that is subsequently transformed in conjunctive normal form (CNF). On the contrary, Z3 provides an API giving access to its built-in AST. When checking real-time models, an additional data structure is needed to encode real-time. Difference bound matrix (DBM) is an implementation of clock zones – an established data structure for real-time – included in UPPAAL [9]. As for stochastic systems, we use Algebraic Decision Diagrams (ADDs) from the CUDD library as a combined representation of variability and probability function [22].

4. ARCHITECTURE

The architecture of ProVeLines is organized into modules, i.e., layers of abstraction. Each module is almost independent and makes use of other modules through narrow interfaces providing only the required operations. The actual implementation of different modules are thus *loosely coupled* since only the signature of the public methods is exposed. Leaning on the object-oriented programming paradigm, it benefits from a clean separation of concerns, a reuse of common functionalities and data types, and it provides facilities to efficiently manage the variability and the extensibility of the tool.

Figure 4 shows the top-level modules of ProVeLines, i.e. the modules that constitute the application programming interface (API) of the model checker. This architecture makes

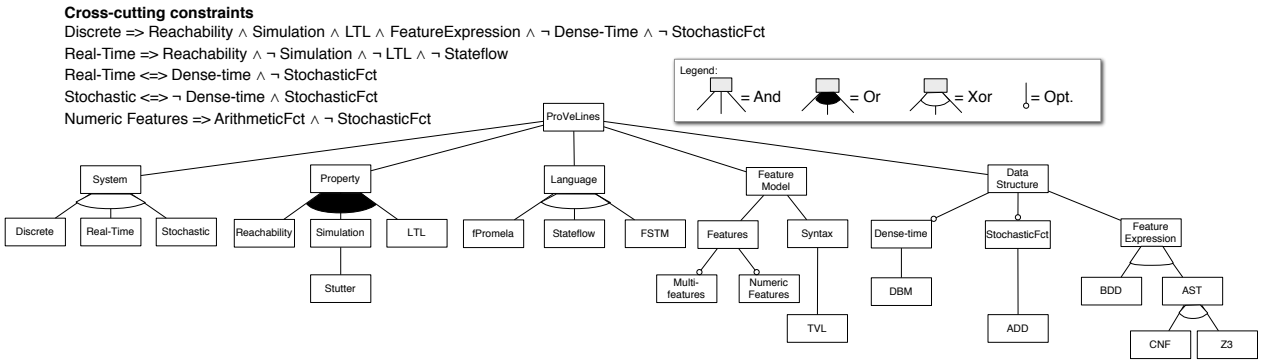

Figure 3: The feature model representing ProVeLines' variability.

ProVeLines extensible with various implementations and additional functionalities. The architecture consists of six top-level modules, each containing smaller modules.

The module named *core* implements the FTS model-checking capabilities. Its most critical part is the *Checker* abstract data type (ADT), inside the *checker* module, which encapsulates the verification algorithms. As shown by the arrows on the diagram, this module depends on the three others. A module depends on another one when it uses at least one of its exposed interfaces. Interfaces define the programming contract of the ADTs they represent. This facilitates their uses by other ADTs, and allows one to implement them independently from the other ADTs.

4.1 Input Language

An important requirement for our new architecture is that existing input languages should be easily customizable, and the addition of new languages should not require modifications in the other components. In particular, input languages and verification algorithms should be agnostic on their mutual implementation. To achieve this, we designed a generic application programming interface (API) named *FTS*, which provides the minimal set of methods needed by an algorithm to perform a verification. Incorporating a new language then comes down to implementing these interfaces. The API itself is actually an abstract, automata-based representation of FTS, and includes methods to access the constructs required by the algorithms. The algorithm thus only has to know the methods provided by the interface in order to perform verifications.

A first abstract data type (ADT) of the input language API is *State*, which represents individual states of the FTS. The ADT *Transition*, represents a transition between two states. As in FTS, transitions are decorated with a feature expression. These are also encapsulated in an ADT (see Section 4.3), which makes an input language independent from the implementation of feature expressions. The last interface is named *FTS*. It consists of a mutable ADT with operations such as getting the current state, computing the transitions available from this state, and executing one of the returned previous states. The mutability of the ADT allows for a more efficient generation of FTS by avoiding the creation and the destruction of many objects, which would result in a lot of memory management work.

Defining the semantics of any input language in terms of FTS allows the reuse of any algorithm designed to verify this formalism. It is also possible to implement round-tripping, *e.g.* to link a violation returned by the checker back to the original model. This capability must be supported by the implementation of the FTS interfaces. The difficulty of this task, and more generally of the implementation of the interface, depends on the complexity of the language itself. In particular, the constructs and data structures offered by this language impact the size of the produced FTS. It may also happen that knowing specific features of the language can lead to optimizations in the verifications. For instance, one can support parallelism by computing parallel compositions in the implementation of the FTS interface. However, heuristics like partial-order reduction requires to access information about the parallel constituents. In this case, one has to make the FTS API evolve. Yet, it is essential to maintain backward-compatibility so as to avoid modifying the algorithms, which depend on this API.

4.2 Properties

The input language API alone provides the necessary methods to compute the set of reachable states in a given model. General properties like deadlock freedom can thus be checked. However, in order to specify more specific properties, it is necessary to provide bridges to temporal logics able to express them. Accordingly, the input language API is extended to offer additional interfaces to represent formulae of a given logic. For instance, we designed an API for Büchi automata, the type of automaton into which Linear Temporal Logic (LTL) formulae are typically transformed to perform verification. The included interfaces are *PropertyState*, *PropertyTransition*, and *PropertyAutomaton*. *PropertyStates* represents states of a Büchi automaton and describes whether this state is accepting or not. *PropertyTransition* and *PropertyAutomaton* are similar to their *FTS* counterparts; they allow the verification algorithms to manipulate the automaton.

If properties are expressed in a logic that cannot be transformed into a Büchi automaton, the existing API is not sufficient anymore. In this case, one has to define additional interfaces providing the minimal set of operations needed to manipulate the new type of property. Then a new variation point has to be created in order to link the appropriate interface according to the desired logic.

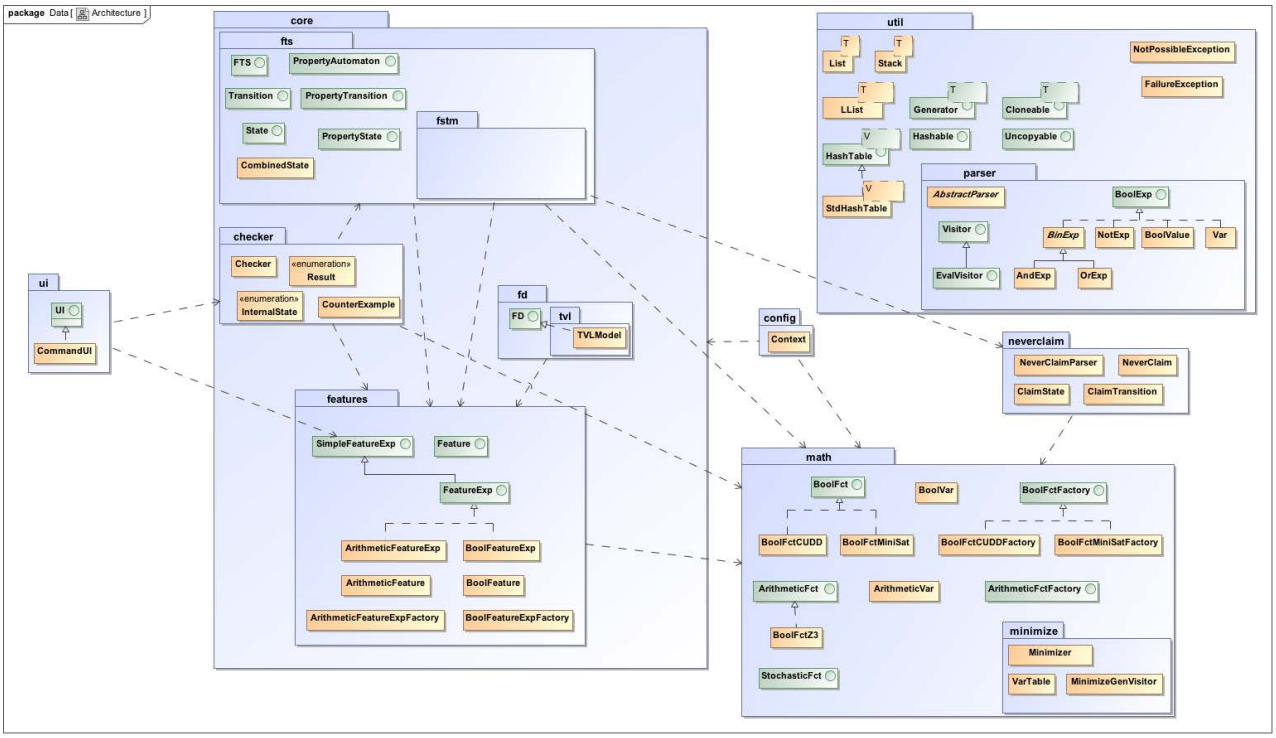

Figure 4: ProVeLines API

4.3 Feature Expressions

As previously mentioned, several data structures can implement feature expressions. We thus provide a top-level interface for feature expressions, i.e., the *FeatureExp* interface. The list of provided methods includes the computation of conjunction, disjunction, and negation, as well as checking satisfiability, validity, implication, and equivalence. Other ADTs further refine this interface, i.e. *BoolFeatureExp* for Boolean feature expressions, and *ArithmeticFeatureExp* for numeric feature expressions [23]. This refinement is achieved by using established object-oriented mechanisms such as inheritance and composition. This hierarchy is carefully studied as the verification algorithms only need to know about feature expressions whereas the input language API may have to know about Boolean feature expressions or arithmetic feature expressions for instance.

4.4 Feature Models

In order to avoid unneeded computations, ProVeLines only verifies the SPL products that are valid according to a given feature model. Feature models have numerous concrete syntaxes, including graphical [34, 39] and textual (see, e.g., [7, 8, 10, 14]). In order to remain independent from the used concrete syntax, we also defined a feature model API. It consists of a single and thin interface with only one method that returns a feature expression representing the underlying feature model. By doing so, we allow any feature-modelling language to be easily plugged in our tool. The corresponding feature expression can then be used during verification according to different strategies [16], which affect the performance of the algorithms and the readability of the results.

4.5 Model-Checking Algorithms

The verification algorithms are encapsulated in a *Checker* ADT. The encapsulation and delegation of work ensure a low coupling between the algorithms and the resources (the formalism, properties, and feature model) they depend on. Thanks to that, one can focus on the design and the optimisation of the algorithm regardless of the structure of its input. This also facilitates the implementation of new verification methods outside the model-checking realm, which may reuse any of the aforementioned APIs. Given that no other module depends on this one, developers are completely free regarding its concrete purpose and its implementation. This means that one can implement any SPL verification procedure, as long as it is based on FTS or one of its extension.

5. WIRING THE MODULES

The architecture described above is sufficiently generic to be derived into a large range of verification tools. The different variants of ProVeLines [24] implement all the aforementioned interfaces into concrete classes that are subsequently combined to form a complete model checker. In this section, we describe the mechanisms we use to link all the modules together.

Given the plethora of existing and future alternative implementations provided in ProVeLines, mechanisms are needed to select the desired modules, that is, to derive a precise variant of the model checker. The desired variant is chosen by means of a configuration file read by ProVeLines at start-up. Leaving this file unchanged makes the configuration persistent across multiple executions of the model

checker. The configuration file can be edited manually or via the support of a dedicated tool. For flexibility purpose, we ensured that both the format and the edition method are independent from the rest of the tool. In particular, the way modules are selected according to the chosen configuration is not impacted by the configuration process.

The different modules are tied together by using inversion of control strategies (also known as dependency injection [26]), service locator [1], as well as other design patterns (e.g., factory and abstract factory [27]). Basically, the *context* of the application is initialized using the specified configuration. The context holds a given configuration of the model checker, and is in charge of instantiating the different APIs according to the selected features. Module dependencies are injected using constructor injection, whereas the appropriate instances of the ADTs are created using a lookup mechanism, often from a factory that is itself injected during construction. These mechanisms allow the variability to be efficiently managed, modular, extensible, and easily testable at the implementation level. Like the previous design decisions, it helps maintaining a clean separation of concerns and a loosely coupled architecture.

Moreover, although the implementations of the different modules are compiled together, it is possible to load a new implementation dynamically without recompiling the whole product line. This provides yet more flexibility in case of extension.

6. PLUGGING A NEW LANGUAGE IN PROVELINES: A GUIDED EXAMPLE

In order to illustrate the benefits of our extensible architecture, we describe step by step the process for building a new language into ProVeLines, and thus for creating a new model checker. For the purpose of this guided example, we define a basic textual language based on a subset of statecharts. It is equipped with constructs such as states, transitions, and transition guards that consist of a feature expression, conditions over variables, or both. We also define a language for specifying LTL formulae based on the property specification patterns [25]. Figure 5 illustrates this sample language through a small excerpt equivalent to the FTS of Figure 1. This figure shows that, when the system is in state `running`, as long as there is no danger it remains therein. On event `danger`, the motor goes to state `stopped` iff the feature `Alarm` is present. Otherwise it remains in state `running` as indicated by the self-transition with the feature expression `!(Alarm)`.

The starting point is to choose between reusing a standard modelling language and defining a domain specific language that fits the needs and habits of the target users. Building a new language implies the definition of a meta-model or a grammar. In any case, we have to develop or reuse a parser to extract a model from the file encoding it. Then we add this parser to ProVeLines as a separate and independent module. ProVeLines also provides a stand-alone API meant to facilitate the development of parsers for grammars of small-to-medium complexity. Once a parser is developed, we have to implement methods that create an abstract syntax model out of a given file. Figure 6 shows the abstract model of our sample language.

The next step consists in implementing the small set of interfaces that made up the input language API (see Section 4.1). Formally, this means that we define the semantics of the language in terms of FTS. Thereby, we create a bridge between the aforementioned abstract syntax model and the FTS formalism. Since the behavioural language is aware of variability, the translation has to maintain a link between the FTS transitions and the products able to execute them. Regardless how variability is represented in the (abstract) syntax, we must encode it as feature expression. As for the actual implementation, we can either reuse those built in ProVeLines or design new implementations. As far as our sample language is concerned, feature expressions are reduced to Boolean formulae over the features. We can thus reuse one of the existing solutions, e.g. the one based on BDDs as implemented in the CUDD library. We can then create an implementation of each of the interfaces of the *FTS* API and make them encapsulate one or more data types of the abstract model of the language.

```
start running {
    on safe { go running }
    on danger {
        featureExp !(Alarm)
        go running
    }
    on danger {
        featureExp Alarm
        go stopped
    }
}

state stopped {
    on safe { go running }
    on danger { go stopped }
}
```

Figure 5: Example of a sample language

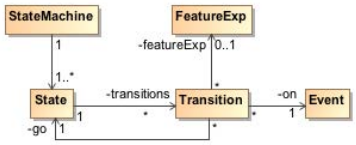

Figure 6: Abstract model of the sample language

For more complex languages, an intermediate representation may be needed. For instance, we might have an abstract syntax tree (AST) built by the parser, and an implementation of the Input Language API which does not depend on this AST. The AST must then be analyzed in order to instantiate the data types of the input language API. This is common programming and is usually quite straightforward with a good design. There are three main ways of traversing an AST: the *interpreter pattern*, the *procedural approach*, or the *visitor pattern* [35]. All these approaches have different characteristics, which we do not discuss here.

Finally the context of ProVeLines has to be updated in order to recognize this new input language, which constitutes a new feature. Here several strategies are available. A first is to link the new implementation to a new feature that is read from the provided configuration file at runtime. A second solution, that can be combined with the first one, is to extend ProVeLines' configuration process to automatically detect the language based on the input file extension. As a

third alternative, ProVeLines supports dynamic loading of external libraries. The advantage of that method is that the code of the context is not changed and thus ProVeLines does not have to be re-compiled. We can then compile the new code in isolation, or with the needed modules if those have not been compiled yet.

7. RELATED WORK

Several SPL verification tools have been developed in the past years. In [17], we extend the NuSMV model checker [12] with the capability to model check SPL whose behaviour is specified in the fSMV language [37]. The resulting tool uses the fully symbolic FTS algorithm presented in [17]. SNIP [15] is ProVeLines' legitimate ancestor; it implements semi-symbolic algorithms where the state space of the system is represented explicitly, whereas variability is symbolically encoded as feature expressions. Its input language is the first variant of Promela we defined. However, it is limited to the verification of LTL formulae on discrete FTS. The previous version of ProVeLines [24] has almost the same functionalities as the one presented in this paper; only a few additional extensions are missing. Its extensibility is limited, though, especially when it comes to implementing new input languages.

There also exist verification tools that are not based on FTS. Ter Beek *et al.* designed a model checker based on modal transition systems and an ad-hoc logic that has similar capabilities as SNIP [41]. SPLVerifier [3] is a verification tool able to detect feature interactions in SPL coded in C or Java. The problem it tackles is different from ours, as we are interested in verifying temporal properties.

PAT3 [36] is a framework for building single-system model checkers. Like us, they facilitate this kind of development by providing an extensible architecture together with a set of APIs. A solution alternative to ours would be to benefit from the platform offered in PAT3.

CPAChecker [11] is a software verification and analysis platform for C programs. It is based on the concept of configurable program analysis, which allows one to express model-checking and program-analysis problems in a single formalism. Like PAT3, CPAChecker offers an extensible architecture that facilitates new development.

Since variability is known to be an invasive concept, supporting it in PAT3 or CPAChecker *directly* would require us to reimplement all their modules and even extend the interface of their APIs. We estimated that it is more beneficial to rely on an SPL-specific tool, *viz.* ProVeLines, and to increase its extensibility rather than incorporating variability into a platform that is not intended for that purpose.

8. CONCLUSIONS

Product-line model checking has reached a sufficient maturity level to be confronted to real-world cases. The next challenge to face is to convince engineers that they can actually use SPL model checkers without having a deep knowledge of the underlying theory. ProVeLines is now ready to be put in use on industrial and real-world cases. Its architecture facilitates the integration of existing modelling languages as well as new languages specifically designed for a given company. The low coupling between input and algorithms permits engineers to use, extend, and tailor ProVeLines even if they do not hold any expertise in model checking. More generally, the separation of concerns that drove ProVeLines development yield numerous facilities that can be used independently and in many contexts. Thereby, we hope that our new toolset will reach many researchers and practitioners.

9. REFERENCES

[1] D. Alur, J. Cupri, and D. Malks. *Core J2EE Patterns: Best Practices and Design*. Prentice Hall International, 2. a. edition, 2003.

[2] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Inf. Comput.*, 104:2–34, May 1993.

[3] 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.

[4] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for product-line verification: case studies and experiments. In *ICSE'13*, pages 482–491, 2013.

[5] 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.

[6] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[7] D. S. Batory. Feature models, grammars, and propositional formulas. In *SPLC*, pages 7–20, 2005.

[8] D. Benavides, S. Segura, P. Trinidad, and A. R. Cortés. Fama: Tooling a framework for the automated analysis of feature models. In *VaMoS'07*, pages 129–134, 2007.

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

[10] D. Beuche. Modeling and building software product lines with pure: :variants. In *SPLC'08*, page 358, 2008.

[11] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In *CAV '11*, pages 184–190, 2011.

[12] 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. Springer, July 2002.

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

[14] 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.

[15] 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.

[16] 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*, pages 1069–1089, 2013.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] M. Cordy, P. Heymans, P.-Y. Schobbens, A. M. Sharifloo, C. Ghezzi, and A. Legay. Verification for reliable product lines. *arXiv preprint arXiv:1311.1343*, 2013.

[23] 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.

[24] M. Cordy, P.-Y. Schobbens, P. Heymans, and A. Legay. Provelines: A product-line of verifiers for software product lines. In *SPLC'13, vol. 2*, pages 141–146. ACM, 2013.

[25] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In *Proceedings of the second workshop on Formal methods in software practice*, FMSP '98, pages 7–15, New York, NY, USA, 1998. ACM.

[26] M. Fowler. Inversion of control containers and the dependency injection pattern, Jan. 2004. http://martinfowler.com/articles/injection.html.

[27] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[28] 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.

[29] A. Haber, C. Kolassa, P. Manhart, P. M. S. Nazari, B. Rumpe, and I. Schaefer. First-class variability modeling in matlab/simulink. In *VaMoS '13*, pages 4:1–4:8, New York, NY, USA, 2013. ACM.

[30] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6:102–111, 1994.

[31] D. Harel. Statecharts: A visual formalism for complex systems. *SCP*, 8(3):231–274, June 1987.

[32] G. Holzmann. *SPIN model checker, the: primer and reference manual*. Addison-Wesley Professional, 2004.

[33] A. Jeanjot. High-level modelling and formal semantics of product-line behaviour. Master's thesis, University of Namur, Belgium, 2013.

[34] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[35] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 2000.

[36] Y. Liu, J. Sun, and J. S. Dong. Pat 3: An extensible architecture for building multi-domain model checkers. In *Proceedings of the 2011 IEEE 22nd International Symposium on Software Reliability Engineering*, ISSRE '11, pages 190–199, Washington, DC, USA, 2011. IEEE Computer Society.

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

[38] A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.

[39] 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.

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

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