Recommendations for Improving the Usability of Formal Methods for Product Lines

Joanne M. Atlee, Sandy Beidu, Nancy A. Day, Fathiyeh Faghih, Pourya Shaker
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Canada
{jmatlee,sbeidu,nday,ffaghihe,p2shaker}@uwaterloo.ca

***Abstract*—While there may be general agreement on what it means for a formal method to be *usable* (e.g., ease of modelling, automated and scalable analysis), there is no consensus in the software-engineering or formal-methods communities on what strategies lead to more usable formalisms. In this paper, we aim to raise discussion around such strategies by proposing fourteen concrete recommendations for achieving practical formal methods. Our recommendations apply to research in formal modelling, automated analysis, and automated transformation (e.g., transforming a model into the input to an analysis tool). Our recommendations focus on formal methods for functional requirements of software product lines, as per our experience in this area as part of a research project in collaboration with an automotive manufacturer; however, most of the recommendations apply to formal methods in general. We also provide a brief overview of a formal modelling language and an under-development tool chain that realizes our recommendations.**

***Index Terms*—formal methods, modelling, analysis, usability, requirements, software product lines**

I. Introduction

Software-engineering researchers who study formal methods (hereafter called *SE/FM researchers*) are interested in transferring formal-methods technologies to industrial software development. They are cognizant of the user experience in employing formal methods, and they investigate research problems that aim to make formal methods more usable or more scalable. At the very least, they consider how to improve the usability and scalability of any technologies that they develop.

When SE/FM researchers write about the *usability* of formal methods, they typically refer to the following objectives:

- *Ease the modelling and evolution of software models.* The most commonly stated barrier to the adoption of formal methods is that the formalisms are inherently difficult to use. The tasks of creating formal representations of software (and keeping them up-to-date) is time consuming and error prone. In some cases, the formalisms require expertise or mathematical aptitude that is uncommon among software practitioners. Formal notations and processes should be designed with the intent to improve the productivity of both the specifiers and reviewers of formal models.
- *Maximize automation.* One means of drastically improving developer productivity is by automating time-consuming tasks. If the tasks are error prone, automation also improves the quality of models and software.
- *Improve the scalability of automated verification.* Probably the second most commonly stated barrier to the adoption of formal methods is that automated verification techniques do not scale to industrial-size models. Addressing this goal includes developing new methods for downscaling the verification problem, as well as integrating techniques that optimize the verification.
- *Improve confidence in analysis results.* Given that automating a formal method is itself a software-development problem, software practitioners may question whether the automation has been implemented correctly. They may lack confidence in an automated analyzer or in any automated transformation of formal descriptions. Improving an analyzer's feedback (e.g., visualization of examples and counterexamples) can help, if users can easily relate the feedback to the original model.

Although SE/FM researchers seem to agree on the above objectives, they pursue very different strategies, suggesting a lack of consensus on how best to achieve these goals.

The purpose of this paper is to promote discussion about how best to improve usability of formal techniques. To initiate discussion, we propose fourteen concrete recommendations for making progress on the above objectives. Our recommendations are based on twenty years of research experience in trying to apply, adapt, and extend formal modelling and analysis techniques to the requirements and high-level designs of large-scale software systems, such as flight-control systems, air-traffic–control systems, product lines of telephony features, and product lines of automotive features. We focus on requirements-level models, because they are smaller to analyze and because analysis of them can reveal errors early in the development process, when errors are cheaper to fix. Some of our recommendations are specific to the modelling and analysis of behavioural requirements or product-line models, but many recommendations simply aim to make formal methods in general more usable.

In the rest of the paper, we present our recommendations and the rationale behind them, and we overview our current work on realizing these recommendations in a tool chain that supports the modelling and analysis of requirements for software product lines.

II. Recommendations

Below, we present fourteen recommendations for making formal methods more usable. The recommendations are grouped according to whether they affect research directions in modelling languages, automated analyses, or automated model transformations.

A. Modelling Recommendations

This section describes our recommendations for easing formal modelling and model evolution, with an emphasis on requirements modelling for software product lines.

1) Build on standard modelling languages: There is a major disconnect between the modelling languages developed by the software-engineering community and those developed by formal-methods researchers. On the one hand, software-engineering modelling languages are designed for direct use by practitioners: they provide a rich set of constructs to enable natural descriptions of a variety of systems, from a variety of view points. Practitioners are trained in the use of software-engineering modelling languages. However, many such languages lack precision and so their models are not amenable to analysis. On the other hand, many formal-methods modelling languages can be thought of as *abstractions* of software-engineering modelling languages that are designed to support reasoning and automated analysis. A viable approach towards practical formal methods is to add precision to standard software-engineering modelling languages, such as the UML and feature diagrams. Making such languages precise enables translation to formal modelling languages for the purpose of analysis (e.g., [32]). Following this approach, we recommend adapting standard software-engineering modelling languages to support precise product-line modelling, rather than inventing a new language for this purpose.

2) Support configurable semantics: Modelling notations can have variable semantics. Semantic variants emerge from applying the language to systems that are naturally described using different semantics. For example, statechart variants [43] differ in several variation points, including the semantics of concurrency (e.g., interleaving vs. true concurrency), the persistence of events (e.g., one vs. multiple execution steps after the event is generated), and the variable values used to evaluate expressions (e.g., the current values vs. the values from the last stable state). To make an informed choice of language variant for a specific modelling task, the practitioner must be aware of the language's semantic variations, as well as the criteria for choosing amongst them. Unfortunately, a practitioner's choice of language variant is typically limited by the available tools (which support only a small subset of the variants). As such, a modeller is often forced to use modelling hacks to simulate missing semantic options (e.g., modelling the persistence of events in a language that does not support it, by introducing auxiliary variables to record the latched events).

We recommend employing and supporting *semantically configurable* modelling languages. Modelling tools should provide facilities that allow a modeller to choose among the semantic options of the language. Furthermore, there should be prescriptive guidance on how to choose among the provided semantic options. One can use semantic configurability to impose problem-specific formal semantics to languages that do not have specific semantics (e.g., the UML).

3) Support (feature) modularity: A fundamental approach for managing complexity in software development is to decompose the system along separate concerns. In software product lines, feature decomposition is the main criterion of separation [26], where a *feature* is a "coherent and identifiable bundle of system functionality" [40]. In product-line development, a family of related software products (e.g., smart phones, automobile models) shares a common set of mandatory features, and products are differentiated by their variable (optional or alternative) features [44]. Ideally, all software artifacts (e.g., requirements models, design, code) are decomposed into *feature modules*.

Although features are modelled as separate feature modules, the modeller will eventually want to visualize and (manually or automatically) analyze *feature combinations* corresponding to products of the product line. To do so, the modeller must derive models of feature combinations by composing feature modules using one of two approaches: (1) composing a *valid* selection of feature modules to obtain a model of a particular product; (2) composing all of the feature modules to obtain a model of the whole product line (representing *all* products). The second approach does not preclude the need for the first approach: depending on how individual products are distinguished in a model of the whole product line (e.g., annotating transitions or actions with presence conditions of the relevant optional features), it may not be easy to visualize models of individual products from the product-line model. We recommend automating both approaches of composing feature modules. As explained in II.B, a composed model of the whole product line, with all of its variabilities, enables more efficient analyses of the product line's set of products.

4) Support modelling feature enhancements as differences: A product line is primarily evolved by adding new features. Sometimes, the purpose of a new feature is to *enhance* an existing feature. As an example from the automotive domain, various advanced cruise-control features enhance traditional cruise control with additional criteria for maintaining the vehicle's speed (e.g., distance from the vehicle ahead, the speed limit). It is natural to model such features in terms of their *differences* from the enhanced features (e.g., [1], [4], [24]). In this approach, the model of the enhanced feature is *reused* as the context for expressing the new feature. In addition to the benefits of reuse, this approach has the benefit of making the task of modelling (at least some features) smaller and more focused on the new feature's essential enhancements – thereby, easing some modelling and evolution tasks.

5) Support commutative feature composition: In some cases, a new feature is designed to *modify* the behaviours of existing features. For example, adaptive cruise control is designed to override traditional cruise control's computation of the vehicle's speed, when there is a slower vehicle ahead.

It is common to realize such *intended interactions* by imposing a *total order* in which features are composed (e.g., [1], [4], [23]), so that a subsequently-composed feature can modify the behaviours of previously-composed (lower priority) features. Such approaches have the benefit of resolving unknown conflicts between any pair of features through priority; however, such resolutions are not explicit and may not be desired. Furthermore, because feature priority is realized through composition order, the composition is *non-commutative* – which, as explained in Section II.B, adds significantly to the analysis task.

To avoid the problems of the above approach, we recommend – perhaps controversially – that feature composition be commutative. Note that such a decision precludes using composition order to (implicitly) realize feature priorities. Instead, we recommend modelling intended interactions overtly, for example by overtly documenting feature priority or, better, explicitly specifying overrides of particular feature transitions or feature actions. With this approach, the only resolutions performed are those that are explicitly modelled. Remaining interactions manifest themselves as ambiguities, nondeterminism, or inconsistencies to be detected in analysis.

Note that this recommendation is distinct from Recommendation 4: a new feature's intended interactions with existing features can be realized using composition order or not, regardless of whether the new feature is modelled as differences from existing features or as a standalone feature.

6) Support additive evolution: Evolving a formal model to incorporate new functionality is generally a non-trivial task. The modeller must be wary of the syntactic and semantic impacts that a change, made to one part of the model, has on other parts of the model. We promote the *additive evolution* of formal models: that is, new functionality should ideally only add elements to the model. This eases the task of model evolution (at least with respect to evolution for the purpose of enhancement). Moreover, by not changing or removing existing elements, we avoid dangling references to the changed/removed elements.

Additive evolution is particularly effective for product-line modelling, where evolutions typically add *optional* features. Even if a new optional feature overrides existing behaviours, it is important to keep the original behaviours in the model (rather than removing them), because they become the product's behaviour whenever the optional feature is not present in the product.

Note that this recommendation complements Recommendation 4 by requiring that, in modelling a new feature's requirements as differences from existing features, the differences be additive (i.e., extensions of existing features), even when expressing overrides.

B. Analysis Recommendations

This sections presents a number of recommendations for scaling up formal-verification techniques to large-scale systems and software product lines.

7) (Normally) Adopt and extend state-of-the-art analysis tools: A common practice among SE/FM researchers is to develop new tools to demonstrate novel ideas and techniques. While we recognize there is a place for building one's own tools, for example as proofs of concept, once the concepts have been proven, they should be transferred to state-of-the-art open-source tools. The danger of developing new closed-source tools is that they have a relatively short lifespan: the student-developers graduate, or research funding ends. Instead, we believe that the default should be to adopt and extend a small collection of powerful analysis tools. Examples of open-source analyzers that implement and integrate state-of-the-art analyses and optimizations include SPIN [22], NuSMV [10], PVS [34], ACL2 [27], CPAchecker [8] and JavaPathFinder [42]. These tools are supported by a community of users and developers, and are continuously improving.

For instance, in the case of product-line analysis, several special-purpose tools have been built from scratch (e.g. [12], [37]). These tools may not develop a large user base and do not benefit from all of the state-space optimizations being invented by the formal-methods community. An example where a state-of-the-art tool was extended was the work by Classen et al. [13] in which the NuSMV model checker was extended for product-line analysis.

Of course we recognize that not all analysis techniques can be integrated and there are times when new technology is disruptive and warrants the development of new tools (e.g. BDD-based model checking, SAT-based model checking). However, by embedding as many analysis and optimization techniques as possible in the same tool, the SE/FM community stands a chance of addressing scalability problems.

8) Analyze the whole product line instead of individual products: Analysis of a product line can be achieved by either (1) generating the individual product variants and analyzing each product separately or (2) analyzing the whole product line and all of its variabilities as a single model. While it is possible to generate automatically all product variants (e.g., [7], [35]) and check them in isolation, this approach may not scale to large product lines because the number of products to analyze is exponential in the number of optional features. Moreover, it has been shown that model checking the whole product-line model [3], [13], [21], [38] is more efficient because analysis can exploit the commonalities among different products [14], [39]. We, therefore, recommend product-line analysis over product-based analysis. To further improve the efficiency of the analysis, the product-line model should include information about valid feature configurations, so that the analyzer examines only valid product variants.

To ensure that the feedback from product-line analysis is comparable to the feedback from product-based analysis, it is not enough for the analyzer to report simply whether a property is satisfied or not. If a property can be violated, the analyzer should report all product variants in which the property can be violated.

2 Revisited) Support commutative feature composition: If feature composition is not commutative, then the order of composition can affect analysis results: features may interact when composed in one order but not in another order. This affects the performance of the analysis because it is likely that multiple feature orderings need to be examined. In approaches that rely on ordered composition, intended interactions will define a partial order among features. However, to analyze a product, the partial order must be extended to a total composition order – preferably one that introduces no unintended interactions. The analysis must check candidate total orderings until a suitable one is found. To search thoroughly for all possible feature interactions, the analysis would need to check all total orderings that extend the designed partial order. The impact on the cost of the analysis is particularly relevant to product-line analysis, which already examines all valid combinations of features; we do not want to check multiple orderings of all valid combinations of features.

9) Automate the generation of correctness properties: Even with the continued advancements in automated analysis, the user must still specify the correctness property to be checked. This raises two issues: (1) identifying a complete and effective set of correctness properties (including detection of vacuity [6]) and (2) correctly expressing the properties in a formal property language such as temporal logic [18].

Correctness properties can broadly be classified as *application-dependent* (e.g. safety, security) or *application-independent* (e.g deadlock, livelock, nondeterminism). There has been much research that investigates how to ease the task of expressing application-dependent properties including graphical/visual notations [15], [28], syntactic sugar [5] and specification patterns [17]. Research on application-independent properties has focused on identifying or generating properties automatically. For example, the FDR tool from Formal Systems (Europe) comes with a property checker for detecting deadlock, livelock and nondeterminism in CSP models [9]. As a second example, some types of feature interactions (e.g. conflicting assignments [16] or conflicting actions to actuators [25]) can be expressed as properties derived solely or mostly from the features' formal models. We recommend trying to formulate correctness properties such that more of them can be generated automatically or semi-automatically.

C. Transformation Recommendations

The overall purpose of the transformation process is to translate a verification problem into the input language of existing formal-methods tools. Below are concrete recommendations for a transformation process that supports the usability objectives of maximizing automation and improving confidence in the results.

10) Fully automate the transformation process: To maximize automation, the transformation process itself must be fully automatic. It should require no user input regarding abstractions for analysis or compatibility with the destination language. Automated transformation helps to realize the goal of ease of evolution by supporting an iterative analysis process: as errors are detected and corrected or as the product line evolves with the addition of new features, the transformation and analysis steps will be run repeatedly.

11) Interpret analysis results in terms of the source model: To produce useful feedback to the modeller, it must be possible to interpret analysis results in terms of the source model. This places a number of requirements on the transformation process: (1) The transformation should produce a destination model whose set of behaviours is equivalent to that of the source model if the destination language is sufficiently expressive. (2) If any abstractions are employed during transformation, they must be reversible or concretizable when interpreting results. (3) The source and destination models must use the same name spaces. (4) At a more complex level, the source and destination models must have the same execution semantics (e.g., the same order and concurrency of transitions). To achieve matching semantics, the transformation may need to add to the destination model auxiliary variables (e.g., to record history information).

12) Make the transformation tool semantically configurable: A naïve way for the transformation process to support Recommendation 2 (configurable semantics) is to offer a suite of transformation tools, one for each possible semantics. Instead, we recommend developing a single semantically configurable transformation tool. The implementation of such a tool can exploit commonalities in the semantic variations. A second advantage is that users learn and interface with only one transformation tool.

13) Make the transformation tool extensible: As one of our analysis goals is to use existing tools, and potentially multiple analyzers, we recommend that transformation tools be designed and constructed so that they can be easily extended to additional target analysis tools. Such extensibility means identifying likely variability points in the space of input languages to analysis tools (e.g., data types, concurrency semantics, step semantics) and encapsulating these variability points in the transformation tool—in the same way that Recommendation 12 entails encapsulating the semantic variability points in the space of the source modelling languages.

14) Verify the transformation: In order to increase confidence in the analysis results, there should be a means of verifying, at least in part, that the transformation has produced an artifact that is equivalent in behaviour to the source model. This goal is complicated by Recommendation 12 (to support semantically configurable transformation) because the verification must cover a large and potentially growing number of combinations of semantic options. Rather than verify a configurable (possibly evolving) transformation tool, it may be more effective to check that the destination model is correct. A major question is how close such techniques can come to verifying that the produced destination model has an equivalent set of behaviours to the source model. Moreover, transformation verification faces the same issues with respect to scalability as the analysis phase focus.

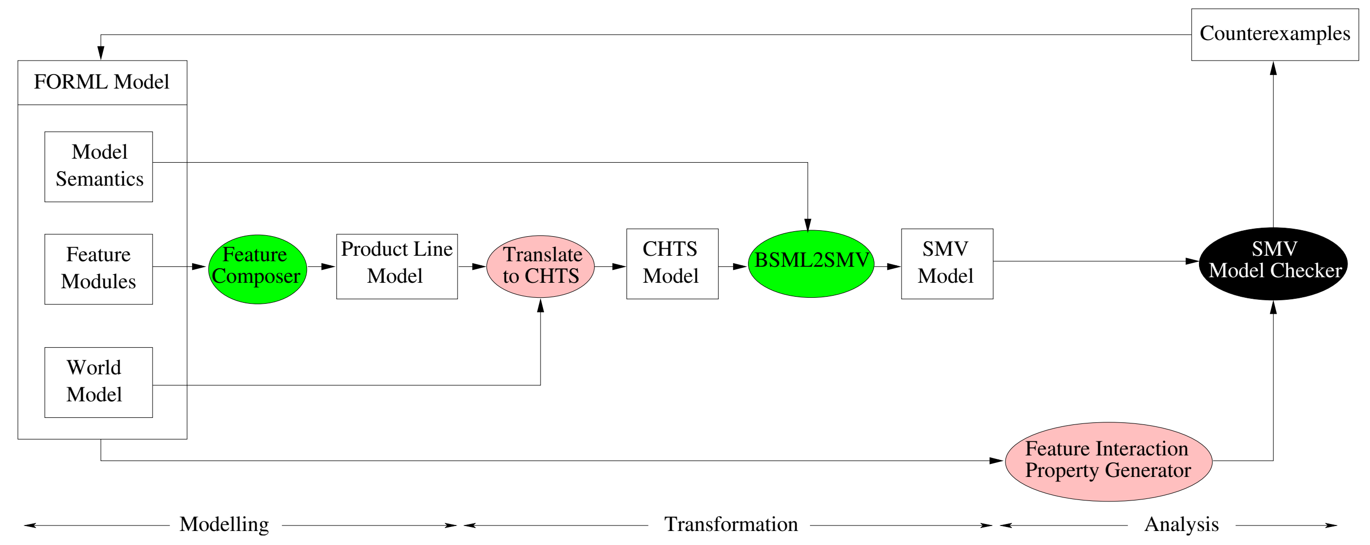

Fig. 1. A tool chain for modelling and analysing product-line requirements. Boxes depict modelling artifacts, green (dark grey) ovals depict tools that we have implemented, pink (light grey) ovals depict tools that we are currently developing, and black ovals depict tools developed by others that we are adopting.

III. Actualizing the Recommendations

The above recommendations are based on our own research in trying to develop modelling and analysis techniques to be used in an industrial setting. In our current work, we are investigating problems in modelling the requirements of automotive software features, and automatically reasoning about the behaviour of feature combinations.

In this section, we briefly overview the current status of our research and describe how different aspects of our work (some of which is completed and some of which is in progress) actualize the various recommendations listed in the previous section. Figure 1 outlines our planned tool chain separated into the modelling, transformation, and analysis phases. While this tool chain may seem relatively long, the invocation of the various tools would be automatic and invisible to the user.

A. Modelling Progress

Two artifacts are produced in the modelling phase of the tool chain in Figure 1: (1) a semantically configured model of the requirements of a software product line that is structured by features (FORML Model); and (2) an integrated model of the product line's requirements (Product Line Model), obtained by automatically composing the models of the individual features (Feature Composer).

FORML Model: The model is expressed in the *feature-oriented requirements modelling language (FORML)* [36]. A FORML Model is a precise model of a product line's requirements, comprising two main views: (1) the World Model, which is an ontology of concepts that describes the *world* in which the product line's members (products) will operate (e.g., objects, events, and other systems in the products' environment); and (2) a set of Feature Modules that separately describe the desired effects of each feature on the world. Some of the key properties of FORML that contribute to the modelling goals in Section II.A are as follows:

- FORML is a UML-like language: the World Model and Feature Modules are based on UML class diagrams and UML state machines, respectively. for example, as in KAOS [41] and Larman's approach [29], class diagrams are used to model a system's world rather than its internal components, and state machines are used to model environmentally observable behaviour as opposed to the behaviour of individual objects.
- FORML supports feature modularity.
- FORML supports commutative feature composition. Intended interactions are specified explicitly as overrides of existing features' transitions or actions (rather than implicitly through ordered composition).
- Adding a new feature requires only additive changes to a FORML Model's Feature Modules – even when the new feature includes intended interactions or is modelled as enhancements to existing features.

FORML has a semantically configurable execution semantics, based on an existing foundation for configurable semantics developed by Esmaeilsabzali et al., called *big-step modelling languages* (BSMLs) [19]. BSMLs is a family of executable behavioural modelling languages (e.g., various statechart variants, process algebras). Each BSML has a syntax that is mappable to a normal-form syntax, called *composed, hierarchical transition system (CHTS)* – an automaton-based language with a rich syntax for describing transition triggers and actions. The semantics of a BSML can be described by selecting from options associated with eight semantic variation points. Each semantic option is characterized in terms of its advantages and disadvantages, as a means of advising modellers on how to define the semantics of a particular BSML. We are working on casting FORML as a subset of the

BSML family, by mapping its syntax to CHTS and identifying the subset of semantic variation points and associated semantic options that are relevant to FORML. This will enable the modeller to choose the Model Semantics of a FORML Model (see Figure 1) in terms of a set of BSML semantic options.

Feature Composer and Product Line Model: We have automated the composition of a set of Feature Modules into a Product Line Model, as denoted by the Feature Composer tool in Figure 1. Our Feature Composer was implemented using the FeatureHouse framework [2]. FeatureHouse provides a generic framework for the structural composition of feature modules using superimposition.

B. Transformation Progress

In our current tool chain, depicted in Figure 1, the transformation step takes as input a Product Line Model (representing a set of features), a World Model and the Model Semantics (representing the semantic choices that have been made), and produces an SMV model to be analyzed.

BSML2SMV*:* We have developed a single, fully automatic, semantically configurable transformation tool called BSML2SMV [20] that takes as input the CHTS normal-form syntax and a set of parameters specifying semantic options (i.e., choices within the configurable semantic space). BSML2SMV supports the entire family of BSML languages [19] and produces models in the SMV language [31]. BSML2SMV builds on work from a previous semantically configurable transformation project [30] that supported a more implementation-oriented semantic space called *template semantics* [33]. BSML2SMV supports the newer semantic framework and produces SMV models whose modularity more closely matches that of the input CHTS model.

The translator BSML2SMV implements most of the recommendations we laid out in Section II.C: it is fully automatic and supports configurable semantics. It produces an SMV model whose level of abstraction and name space are the same as those of the source model. We are currently investigating methods for verifying the correctness of models produced by BSML2SMV. In future work, we will consider extending our transformation tool to support transformation to additional rich target verification languages so that we will have access to more analysis tools.

Translate to CHTS: To use BSML2SMV, we must transform the FORML Product Line Model and World Model into the CHTS normal form syntax. Because CHTS is automata based, the mapping of the Product Line Model (a transition system) is straightforward. The World Model, however, is a data model based on the UML class diagram. Both SMV and CHTS currently support only low-level data types. Therefore, the World Model, which contains rich data types, has to be encoded using the available low-level primitive types, such as arrays and enumerations. We are still experimenting with efficient ways of encoding the World Model.

C. Analysis Progress

Our main goal in analyzing a Product Line Model is to detect unintended interactions among the product line's features. As shown in Figure 1, the input to the analysis phase of our tool chain is an SMV Model representing the entire product line. Properties to detect feature interactions are automatically generated based on the FORML Model of the features' individual requirements. The SMV Model Checker reports for each violated property (representing an unintended feature interaction) the set of products in which the interaction can occur. Below we discuss the progress we have made with regards to automating and optimizing the analysis task.

Feature Interaction Property Generator: To generate automatically correctness properties for detecting feature interactions, we first had to identify the different ways in which feature interactions manifest themselves. Currently, we have identified and formulated correctness criteria for detecting interactions due to nondeterminism or conflicting actions. The Property Generator outputs correctness properties expressed in Computation Tree Logic (CTL) [11]. The properties are generated automatically from the requirements models of the features without any input from the user, which is in line with our goal of fully automating the analysis and improving the usability of the method.

Property generation is complicated by the fact that some feature interactions are intended and desirable, and should not be reported as errors in the model. Since our FORML modeling language supports the explicit modelling of intended interactions, as discussed in Section III.A, we are able to encode properties such that only unintended interactions are reported. This improves the feedback to the user and helps the user to focus only on troubleshooting undesired behaviour.

SMV Model Checker: We are currently using an extended version of the NuSMV model checker [10] that is capable of analyzing product-line models [13]. The model checker takes as input an SMV model representing a product-line and analyzes the full product-line model. The model checker outputs for each violated property a Boolean expression that characterizes all of the feature configurations (or products) that violate the property; it also outputs a counterexample showing how the property can be violated in one of the output products. However, this work builds on BDD-based model checking, which is no longer state-of-the-art for software models, so we are currently exploring other optimized model checkers, to see if they can be extended to analyze product-line models.

IV. Conclusions

We have presented fourteen recommendations of concrete strategies to improve the usability of formal methods. Our recommendations focus on the modelling and analysis of behavioural requirements for software product lines, but many of them generalize to other application domains. We intend for these recommendations to be input to a larger discussion within the SE/FM community on how to improve the usability of formal methods.

References

[1] S. Apel, F. Janda, S. Trujillo, and C. Kästner, "Model superimposition in software product lines," in *Proc. Int. Conf. on Theory and Practice of Model Transformations (ICMT)*, 2009, pp. 4–19.

[2] S. Apel, C. Kästner, and C. Lengauer, "FeatureHouse: Language-independent, automated software composition," in *Proc. Int. Conf. on Software Engineering (ICSE)*, 2009, pp. 221–231.

[3] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer, "Detection of feature interactions using feature-aware verification," in *Proc. Int. Conf. on Automated Software Engineering (ASE)*, 2011, pp. 372–375.

[4] D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," *IEEE Trans. on Software Engineering*, vol. 30, pp. 355–371, 2004.

[5] I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh, "The temporal logic sugar," in *Proceedings of Computer Aided Verification (CAV)*, 2001, pp. 363–367.

[6] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh, "Efficient detection of vacuity in temporal model checking," *Formal Methods in System Design*, vol. 18 (2), pp. 141–163, 2001.

[7] D. Beuche, H. Papajewski, and W. Schröder-Preikschat, "Variability management with feature models," *Science of Computer Programming*, vol. 53, pp. 333–352, 2004.

[8] D. Beyer and M. E. Keremoglu, "CPAchecker: A tool for configurable software verification," in *Proceedings of Computer Aided Verification (CAV)*, 2011, pp. 184–190.

[9] N. Brownlee, "Formal systems (europe) ltd: Failures-divergence refinement – FDR2 user manual," in *Blount MetraTech Corp. Accounting Attributes and Record Formats http://www.ietf.org/rfc/rfc2924.txt*, 2000.

[10] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An opensource tool for symbolic model checking," in *Proceedings of Computer Aided Verification (CAV)*, 2002, pp. 359–364.

[11] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. on Programming Languages and Systems*, vol. 8, pp. 244–263, 1986.

[12] A. Classen, M. Cordy, P. Heymans, A. Legay, and P.-Y. Schobbens, "Model checking software product lines with SNIP," *Journal on Software Tools for Technology Transfer*, vol. 14, no. 5, pp. 589–612, 2012.

[13] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay, "Symbolic model checking of software product lines," in *Proc. Int. Conf. on Software Engineering (ICSE)*, 2011, pp. 321–330.

[14] 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 *Proc. Int. Conf. on Software Engineering (ICSE)*, 2010, pp. 335–344.

[15] A. Del Bimbo, L. Rella, and E. Vicario, "Visual specification of branching time temporal logic," in *Proc. Int. Sym. on Visual Languages*, 1995, pp. 61–68.

[16] D. Dietrich, P. Shaker, J. M. Atlee, D. Rayside, and J. Gorzny, "Feature interaction analysis of the feature-oriented requirements-modelling language using Alloy," in *Proceedings of the MoDELS Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa)*, 2012, pp. 17–22.

[17] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proc. Int. Conf. on Software Engineering (ICSE)*, 1999, pp. 411–420.

[18] E. A. Emerson, "Temporal and modal logic," in *Handbook of Theoretical Computer Science*, 1995, pp. 995–1072.

[19] S. Esmaeilsabzali, N. A. Day, J. M. Atlee, and J. Niu, "Deconstructing the semantics of big-step modelling languages," *Requirements Engineering Journal*, pp. 235–265, 2010.

[20] F. Faghih and N. A. Day, "Mapping big-step modelling languages to SMV," David R. Cheriton School of Computer Science, University of Waterloo, Tech. Rep. CS-2011-29, 2011.

[21] A. Gruler, M. Leucker, and K. D. Scheidemann, "Modeling and model checking software product lines." in *Proc. Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, 2008, pp. 113–131.

[22] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. on Software Engineering*, vol. 23, pp. 279–295, 1997.

[23] M. Jackson and P. Zave, "Distributed feature composition: A virtual architecture for telecommunications services," *IEEE Trans. on Software Engineering*, vol. 24, no. 10, pp. 831–847, 1998.

[24] P. K. Jayaraman, J. Whittle, A. M. Elkhodary, and H. Gomaa, "Model composition in product lines and feature interaction detection using critical pair analysis," in *Proc. Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS)*, 2007, pp. 151–165.

[25] A. L. Juarez-Dominguez, N. A. Day, and J. J. Joyce, "Modelling feature interactions in the automotive domain," in *Proc. of ICSE Workshop on Models in Software Engineering (MiSE)*, 2008, pp. 45–50.

[26] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Carnegie-Mellon University Software Engineering Institute, Tech. Rep. CMU/SEI-90-TR-21, 1990.

[27] M. Kaufmann, J. S. Moore, and O. Boyer, "An industrial strength theorem prover for a logic based on Common Lisp," *IEEE Trans. on Software Engineering*, vol. 23, pp. 203–213, 1997.

[28] G. Kutty, L. K. Dillon, L. E. Moser, P. M. Melliar-Smith, and Y. S. Ramakrishna, "Visual tools for temporal reasoning," *Proc. Int. Sym. on Visual Languages*, pp. 152–159, 1993.

[29] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process (3rd Edition)*. Prentice Hall, 2005.

[30] Y. Lu, J. M. Atlee, N. A. Day, and J. Niu, "Mapping template semantics to SMV," in *Proc. Int. Conf. on Automated Software Engineering (ASE)*, 2004, pp. 320–325.

[31] K. McMillan, "Symbolic model checking: An approach to the state explosion problem," Ph.D. dissertation, School of Computer Science, Carnegie Mellon University, 1992.

[32] W. E. McUmber and B. H. C. Cheng, "A general framework for formalizing UML with formal languages," in *Proc. Int. Conf. on Software Engineering (ICSE)*, 2001, pp. 433–442.

[33] J. Niu, J. M. Atlee, and N. A. Day, "Template semantics for model-based notations," *IEEE Trans. on Software Engineering*, pp. 866–882, 2003.

[34] S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert, "PVS: an experience report," in *Applied Formal Methods—FM-Trends 98*, ser. LNCS, vol. 1641. Springer-Verlag, 1998, pp. 338–345.

[35] G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel, "FeatureIDE: A tool framework for feature-oriented software development," in *Proc. Int. Conf. on Software Engineering (ICSE)*, 2009, pp. 611–614.

[36] P. Shaker, J. M. Atlee, and S. Wang, "A feature-oriented requirements modelling language," in *Proc. Int. Requirements Engineering Conf. (RE)*, 2012, pp. 151–160.

[37] M. H. ter Beek, S. Gnesi, and F. Mazzanti, "Demonstration of a model checker for the analysis of product variability," in *Proc. Int. Software Product Line Conf. (SPLC)*, 2012, pp. 242–245.

[38] M. H. ter Beek, F. Mazzanti, and A. Sulova, "VMC: A tool for product variability analysis," in *Proc. Int. Sym. on Formal Methods (FM)*, ser. LNCS, vol. 7436, 2012, pp. 450–454.

[39] T. Thüm, I. Schaefer, S. Apel, and M. Hentschel, "Family-based deductive verification of software product lines," in *Proc. Int. Conf. on Generative Programming and Component Engineering (GPCE)*, 2012, pp. 11–20.

[40] C. R. Turner, A. Fuggetta, L. Lavazza, and A. L. Wolf, "A conceptual basis for feature engineering," *Journal of Software and Systems*, vol. 49, no. 1, pp. 3–15, 1999.

[41] A. van Lamsweerde, *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[42] W. Visser and P. C. Mehlitz, "Model checking programs with Java PathFinder," in *Proc. SPIN Sym.*, 2005.

[43] M. von der Beeck, "A comparison of statecharts variants," in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, 1994, pp. 128–148.

[44] D. Weiss and R. Lai, *Software Product Line Engineering: A Family Based Development Process*. Addison Wesley, 1999.