A Survey on Modeling Techniques for Formal Behavioral Verification of Software Product Lines

Fabian Benduhn
University of Magdeburg
Magdeburg, Germany
fabian.benduhn@ovgu.de

Thomas Thüm
University of Magdeburg
Magdeburg, Germany
tthuem@ovgu.de

Malte Lochau
TU Darmstadt
Darmstadt, Germany
malte.lochau@es.tu-darmstadt.de

Thomas Leich
METOP GmbH
Magdeburg, Germany
thomas.leich@metop.de

Gunter Saake
University of Magdeburg
Magdeburg, Germany
saake@ovgu.de

ABSTRACT
As software product lines are increasingly used for safety-critical systems, researchers have adapted formal verification techniques such as model checking and theorem proving to cope with compile-time variability. While the focus of the ongoing debate lies on the verification mechanisms itself, it becomes increasingly difficult for researchers to maintain an overview about the various accompanying modeling techniques. We survey existing approaches as a first step towards a unifying view on variability mechanisms in formal modeling techniques for product lines. We illustrate the approaches by means of a running example to illustrate their commonalities and differences.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software/Program Verification—*modeling, variability*; D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms
Design, Languages, Verification

Keywords
Software Product Lines, Variability, Survey, Modeling, Verification

1. INTRODUCTION
Today, software systems must often be developed in a large variety of variants to meet the requirements of different customers. Software product line engineering is a paradigm of software development in which multiple products that share a common set of development artifacts are developed simultaneously [16, 48]. Each product of a product line is considered as a combination of features [3]. A feature is a user-visible characteristic of a software system [33].

VaMoS '15, January 21 - 23 2015, Hildesheim, Germany
Copyright 2015 ACM 978-1-4503-3273-6/15/01...$15.00
http://dx.doi.org/10.1145/2701319.2701332

Software product lines are increasingly used for the development of safety-critical and mission-critical systems in which software errors may have consequences that cannot be tolerated [61, 8]. For single-system engineering, the use of formal specification and verification techniques has emerged as an approach to establish correctness properties [12]. A challenge of ongoing research is to adapt formal verification techniques to cope with the variability of software product lines [59, 54].

In general, a verification technique consists of a formalism to model the behavior of the system, an accompanying formalism to describe desired behavioral properties, and a well-defined mechanism to check the model against the specification. In recent years, researchers have adopted established verification techniques to cope with the variability of product lines [59].

So far, the literature focuses on handling the variability in the verification mechanisms rather than on the modeling and specification formalisms. A number of surveys include an overview of such techniques to some degree, but none of them focuses primarily on formal modeling [54, 1, 31, 59]. In a survey on analysis techniques for product lines that includes formal verification, Thüm et al. identify the need for a survey of the accompanying modeling and specification techniques for future research [59].

In recent years, several approaches to model the behavior of product lines have been proposed. Each of these approaches has been developed rather independently from each other, being tailored towards a specific verification technique. Despite their differences, they share certain common principles regarding the handling of variablity. We give an overview of existing formal, bevavioral modeling techniques for software product lines. We exemplify different mechanisms to cope with variability by means of a running example to emphasize their commonalities. We see this as a first step towards a unifying view on variability mechanisms in formal, behavioral modeling techniques that may help to take advantage of their commonalities. Our results show that many existing approaches share commonalities and can be divided into a small number of categories with individual advantages and disadvantages.

2. BACKGROUND
Similar to single-system verification, a significant part of the modeling techniques for product lines are based on transition systems (TS). Thus, we give a brief introduction to transition systems.

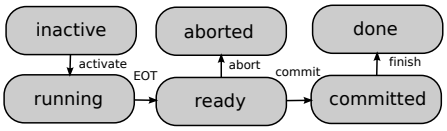

Figure 1: One product from the transaction product line.

TSs are a widely used formalism to model systems [6, 35]. A TS can be seen as a directed graph in which nodes represent program states and edges represent transitions between states. Edges can be labeled with names of actions to describe the behavior related to the execution of the transition. We give the following simple definition adapted from Baier and Katoen [6]:

A (labeled) transition system TS[1] is a tuple $(S, Act, \rightarrow, I)$ where

- S is a set of states,
- Act is a set of actions,
- $\rightarrow \subseteq S \times Act \times S$ is a transition relation,
- $I \subseteq S$ is a set of initial states.

A TS describes all possible executions of a system. An execution starts in some initial state $s_0 \in I$. In each step of execution, the system evolves according to the transition relation. The outgoing transitions of a state determine the next possible steps of execution. Figure 1 shows an example of a TS of an transaction system. A transaction starts in the state *inactive*. After activation it reaches the state *running*, in which it performs operations on the database. When the end of transaction (EOT) is reached, the transaction either gets aborted or committed depending on whether the performed operations preserve the consistency of the database.

So far, we have seen how systems can be modeled with TS. Such TS are often used for model checking [6, 11]. In model checking, the states of a TS are systematically explored to check whether the system fulfills certain properties. For this purpose, it is neccessary to express such properties formally. This is typically done by using special logics with semantics defined over TS such as Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) [6].

3. MODELING TECHNIQUES FOR SOFTWARE PRODUCT LINES

The specific characteristic of product-line models is that multiple products that share large parts of their behavior but differ in others must be developed simultaneously. Thus, modeling techniques that include a notion of variability have been proposed in the literature. In this survey, we focus on modeling techniques for formal, behavioral verification of software product lines. We present annotation-based modeling techniques for product lines in Section 3.1 and composition-based modeling techniques for product lines in Section 3.2.

3.1 Annotation-Based Modeling

In annotation-based modeling techniques, parts of a model are annotated with information about their mapping to products of the product line. The set of products is typically refered to by means of feature expressions, i.e., by logical expression over the set of features. An important class of annotation-based formalisms are variants of TSs, which are often used for model checking of software product lines. Common to the formalisms based on TSs is that variability is encoded by adding specific labels on transitions of a system. When modeling a product line with a TS, each product of a product line differs in the set of reachable states and the set of transitions to reflect behavioral differences. Commonalities between products manifest in transitions that are common to some or even to all products.

[1] TS is called finite if S and Act are finite.

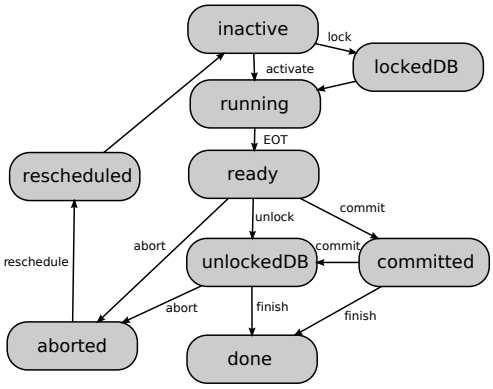

Figure 2: Family-TS of the transaction product line.

Family-TS. The most basic approach to use a TS for the modeling of product-lines, to which we refer to as Family-TS, is to include all transitions of all products into a single TS. This approach has been discussed by Fischbein et al., mainly to motivate the development of further extensions [25]. In this basic approach, no explicit labels to express variability are required. The implicit meaning of a transition T can informally be seen as: There is at least one product that includes transition T.

A Family-TS can be used to reason about a limited set of properties. Consider the case of reachability, i.e., the question whether a certain target state T is reachable from a source state S. If state T is not reachable from state S in the Family-TS, we can conclude that this property holds for all products of the product line. However, if state T is reachable from state S, we cannot conclude that this property holds for any particular product. The reason is that we do not know whether all involved transitions are contained in a given product, i.e., there is no explicit information about the mapping between transitions and products. The Family-TS in Figure 2 contains all transitions from all products of the transaction product line. As there is no transition from state *done* to state *running*, we can conclude that successfully terminated transitions are not rescheduled and executed again.

Modal Transition Systems. To include more information about the mapping between transitions and products of a product line, Fischbein et al. proposed to use Modal Transition Systems (MTS) [25]. An MTS is a TS in which a transition is either mandatory (must) or optional (may). Again, all transitions of all products are included in a single MTS, but the additional labels reveal more information about the mapping between transitions and products. Informally, the meaning of a must-transition is that the transition is contained in all products, and the meaning of a may-transition is that the transition is contained in at least one but not all products.

In Figure 3, we show an MTS for the transaction product line. If we consider the example of reachability, there are now more cases, in which we can reason about properties of products. The

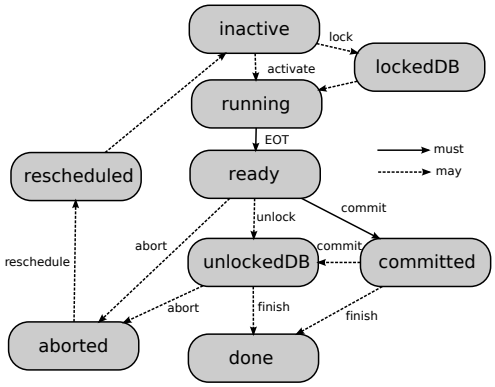

Figure 3: Modal Transition System (MTS) of the transaction product line.

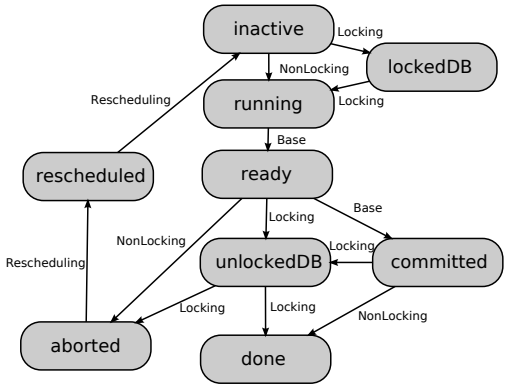

Figure 4: Featured Transition System (FTS) of the transaction product line. The action labels are as usual but have been omitted for simplicity.

path from state *running* to state *committed* contains only must-transitions. We can conclude that state *committed* is reachable from state *running* in all products of the product line. The path between state *lockedDB* and state *committed* includes exactly one may-transition. We can conclude that there exists at least one product in which this is true because a may transition is included in at least one product. However, if there are two or more may-transitions on a path, we cannot conclude anything about the path in products because we do not know whether there exists a product that contains all involved transitions.

A similar formalism, in which MTS are specified based on I/O-automata has been proposed independently [36]. Variability is also achieved by distinguishing between must-transitions and may-transitions. A variant of these variable I/O-automata has been proposed to model individual domain artifacts, i.e., the product line is composed of a set of domain artifacts and each artifact is annotated with variability information [38].

Generalized Extended Modal Transition Systems. Generalized extended modal transition systems (GEMTS) have been proposed as a generalization of MTS [24]. In GEMTS, the modal annotations are applied to hyper-transitions, i.e., transitions to multiple states. Intuitively, this can be seen as a set of transitions with the same source. The informal meaning of a may-transition is that each product contains a certain number n of the annotated transitions. Analogously, a must transition means that each product contains at most n of the annotated transitions. The value of the number n must be defined additionally for each hyper-transition. While GEMTS in this sense are more expressive as MTS, they do not overcome the general limitation that relationships between individual annotations cannot be expressed.

In Family-TS, MTS, and GEMTS, the information about the mapping is not completely embedded into the model, which is their main limitation. For instance, this manifests in whether model checking algorithms are able to pinpoint specific products that violate a given property [14]. In MTS and GEMTS, specific labels express whether a certain transition is contained in all products or not. However, from such a model we cannot conclude the exact set of products that is target of the mapping. To overcome this limitation for analysis purposes, it is possible to enrich the specification technique for properties with additional variability information by using a specific logic [5].

Featured Transition Systems. Featured Transition Systems (FTS) have been proposed by Classen et al. [15]. The initial idea of FTS is to label each transition of an TS with a feature, directly constituting a mapping between the transition and this feature. The meaning of a transition T labeled with feature F can informally be read as: Transition T is contained exactly in those products that contain feature F. In Figure 4, we show an FTS for the transaction product line. As each transition can be labeled with only one feature, we have to introduce a separate feature *NonLocking* as an alternative to feature *Locking*. Otherwise, products containing feature *Locking*, would still contain transitions that circumvent the locking mechanism, e.g., $inactive \rightarrow running$.

The main advantage of FTS is that the information of the mapping between transitions and products is completely revealed [15]. Consider the reachability property. If a path leads from the source state S to the target state T, we can conclude that this path is contained in exactly the products that contain all features that are used as transition labels on this path. However, FTS are limited by the set of possible mappings that can be expressed: Each transition is mapped to only one feature. To overcome this limitation, Classen et al. have extended FTS by allowing transitions to be labeled with feature expressions [19, 18, 14, 22]. In this case, we can replace feature *NonLocking*, by labeling the transitions with the logical negation of feature *Locking*, i.e., with $\neg$*Locking*. By doing so, we associate this transition with all products that do not contain feature *Locking*. In the remainder of the paper, we denote this variant of Featured Transition Systems as FTS.

The language fPromela has been proposed as an input language for FTS for the SNIP model checker, an FTS-based adaption of the SPIN model checker [13]. Figure 5 shows an example of an fPromela model. A transaction is modelled as a process. A special type $features$ is used to declare the set of features as variables. Inside the process $transaction$, guarded statements are used to annotate parts of the code with its mapping to features. In contrast to Promela, these guarded statements are denoted by keyword `gd` rather than keyword `if`. The statements *lockDB*() and *unlockDB*() are only executed if feature *Locking* is present. The SNIP model checker interprets the model as an FTS, i.e., for the feature variables all possible values (true and false) are considered to reason about all products. In an approach similar to FTS, I/O Automata are annotated with a mapping to an orthogonal variability model [37].

```
typedef features {
bool Locking;
bool Rescheduling;
};

features f;

proctype transaction()
{
  ...

  gd :: f.Locking → lockDB();
  performOperations()
  gd :: f.Locking → unlockDB();

  ...

}
```

Figure 5: Example of composition-based Annotations in fPromela

Adaptive Featured Transition Systems. In Dynamic Software Product Lines (DSPLs), the set of features may change at runtime [27] in dependence of the environment. To model DSPLs, Adaptive Featured Transition Systems (A-FTS), have been proposed [17]. In contrast to FTS, A-FTS consider variability of both the system and of its environment by distinguishing between system features and environment features. System features can be fixed or adaptable. Environment features can change over time, and can be observed by the system to initiate reconfigurations. This is achieved by enabling or disabling system features. The main technical difference between FTS and A-FTS, is that the transition relation is given as a function that defines which transitions exist, to which products they belong, and the current state of the system configuration and environment configuration. However, the principle to use feature expressions to map transitions to products is the same.

Featured Timed Automata. Featured Timed Automata (FTA) are a variant of FTS for real time systems [20]. In a real-time SPL, features cannot only change behavior but also make changes to so called timing constraints for actions. In Timed Automata (TA), this is reflected in the use of clock constraints, that define timing properties for actions such as the minimum or maximum time required for execution. FTA extend Timed Automata by support to annotate clock constraints with feature expressions to establish a mapping to features.

State Diagram Variability Analysis Models. State Diagram Variability Analysis (SDVA) models have been proposed as an extension of FTS with means to express hierarchical sub-models to structure the model of a product line [23]. This is achieved by the ability to refine states of a model by defining separate submodels.The semantics of SDVA models are defined over FTS that are derived by flattening the hierarchical structure of the model.

Other Approaches. The idea to annotate state transitions has also been applied to Petri nets [45]. Feature Petri Nets have been proposed to model the behavior of product lines and for context-aware test models [46, 50]. In Feature Petri Nets, the mapping between parts of the model and products is established in the same way as in FTS, i.e., by labeling transitions with feature expressions. As a Petri net can be transformed to a TS, they can be seen as a higher-level modeling formalism that supports a more convenient way of modeling for certain classes of systems. Similarly, process-algebraic approaches have been proposed [60, 51, 30]. PL-CCS is a modeling technique based on Milner's calculus of communicating systems (CCS) that has been enriched with a variant operator as a means to implement variability [51, 30]. Simple hierarchical variability modeling (SHVM) has been proposed in which each variation point of a hierarchical model is mapped to exactly one feature [53]. The limitation of the mapping to target single features provides benefits for compositional verification [53]. In the modeling language FLan, the mapping from model to sets of products is not achieved by means of annotations but given implicitely by treating features as first class entitie [58]. In FLan, variability of processes can be defined by using special operators, e.g., to expres alternativity.

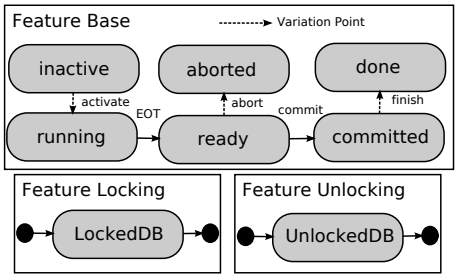

Figure 6: Example of composition-based TS based on Fisler et al.

3.2 Composition-Based Modeling

So far, we have presented annotation-based modeling techniques. In the following, we present composition-based approaches, in which the product line is decomposed into separate modules representing features that can be composed to derive products.

Composition-Based TS. In Figure 6, we exemplify a technique for composition-based modeling based on TS, initially proposed by Fisler et al. [26, 39, 40, 41]. In this technique, a product line is modelled as a base feature with a number of additional feature modules. Intuitively, the base feature can be seen as a TS with special transitions that serve as extension points. At these fixed locations, feature modules can be plugged in to extend the behavior of the model. Each feature module is modelled as a partial TS with special input and output states. In the example, feature *Locking* consists of a single state. In our example, this feature can be plugged in between state *inactive* and state *running* of the base model. A limitation of this approach is that feature modules cannot crosscut the base model, i.e., a feature can only be plugged in as a whole at a given location. Thus, we have to extract the unlocking functionality of feature *Locking* into a separate feature module. Feature *Unlocking* can be plugged in between state *ready* and state *aborted*, or between state *done* and state *committed*.

In Figure 7, we show a similar approach in which feature modules are successively added to a base system. The transitions between states of different features are explicitely defined [42]. Thus, each feature can have multiple incoming and outgoing transitions to arbitrary states of the base system. This allows us to incorporate both state *lockedDB* and state *unlockedDB* in a single module. However, this approach does not allow to replace transitions. Thus, the transition between state *running* and state *ready* of the base feature

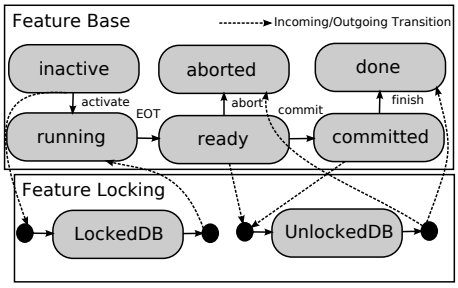

Figure 7: Example of composition-based TS based on Liu et al.

is included in all products. To avoid this, we would have to extract these transitions into a separate feature *NonLocking* that must be included if feature *Locking* is not selected.

Cordy et al. propose a similar composition-based technique, in which the base system is modeled as a TS and feature modules as special transtition systems called TS+ [21]. A TS+ is a transition system with activation conditions that express the states of the base system from which the feature module can be reached, and return conditions that define the set of states into which the feature module may return to the base system.

Finite State Machines with Variability. In research regarding evolution of product lines, Finite State Machines with Variablity (FSMv) have been proposed [44]. FSMv considers for each feature a model of its requirements (FSMr) and a model of its design (FSMd). The purpose of this composition-based approach is to support an open-world assumption of product lines in which previously unknown features may be added to a product line.

Using Conventional Decomposition Mechanisms. A simple technique to composition-based modeling that has been proposed in the literature is to use a conventional modeling language known from single-system engineering and use the existing modularization mechanisms to encapsulate features.

This technique has been investigated in the case of abstract state machines (ASM) [9]. The existing decomposition mechanisms of ASM have been explored regarding their capability to encapsulate features [28, 7]. These development methods apply the idea of stepwise refinement to formal models: an abstract model is successively refined to a more concrete model, and finally to executable code. The existing modularization mechanisms have been shown to be sufficient to modularize features for a model of Java 1.0 and its implementation on the JVM together with correctness proofs [7].

Similarly, a feature-oriented extension of Event-B has been proposed in the literature [28, 56, 49]. In this line of research, the conventional Event-B composition mechanisms have been explored regarding their capabilities to implement and compose features. The difference to the work on ASMs is that a mapping between features and modules has been explicitely considered. We consider both approaches as composition-based modeling techniques, as each feature is encapsulated in a single module. A limitation of this approach is that they support only one dimension of decomposition, which does not allow to express features that crosscut the dominant decomposition structure of a system [57].

For the detection of feature interactions, the modeling language Promela has been used to model the behavior of features [10]. However, these features can only be inserted at fixed locations in the base program.

Feature-Oriented Extensions of Modeling Languages. The idea to extend an existing modeling language with means to modularize features has been applied to modeling techniques. A feature-oriented extension of the formal modeling language Alloy called FeatureAlloy has been proposed by Apel et al. [4]. The main purpose of FeatureAlloy is to bridge the gap between problem space and solution space by means of an abstract model of the product line. A FeatureAlloy model is capabable to encapsulate the behavior of crosscutting features. However, the refinement of models using multiple levels of abstraction as supported by ASM or Event-B has not been considered.

The language fSMV is an extension of the input language for the NuSMV model checker and semantically based on FTS. Notable is that fSMV is a composition-based language with semantics based on an annotation-based formalism. Similarly, Modal Sequence Diagrams (MSD) have been proposed, in which each feature is encapsulated into a separate module and translated to annotation-based SMV model for model checking purposes [29] .

Other Approaches. Delta Modeling is an approach in which a product line is decomposed into a core product and a set of delta modules [43, 32, 52]. Delta modules encapsulate modifications that can be applied to the core product to derive other products of a product line. For instance, DeltaCCS is a delta-oriented extension to Milner's process calculus CCS. Variability is achieved by decomposing the model into a core process and a set of delta modules that encapsulate change directives based on term rewriting semantics [43]. A main difference to the previously discussed feature modules is that delta modules can not only add but also remove parts of a model. A further related line of research in which composition-based models have been proposed is aspect-oriented software development [55, 47, 2, 34]. These approaches also rely on a TS for the base system and further transition systems to model features. The difference to the previously presented approaches is that the considered variability is limited. That means that the aim is to modularize features, but different combinations of features are not considered. Instead, it is typically assumed that all features are included in the product. Despite the fact that these approaches do not focus on variability, it is still possible that different combinations of features are woven into a system to derive different products.

3.3 Discussion

In various lines of research, several formalisms and languages to model the behavior of product lines have been proposed. Despite their differences, these approaches can be divided into two main categories regarding their handling of variability: Annotation-based techniques and composition-based techniques. In the following, we briefly discuss the commonalities and differences of these techniques and their potential limitations.

Annotation-based techniques have in common that the behavior of all products of a product line is modeled in a single model where individual parts are annotated with information about variability. Individual annotation-based modeling techniques differ in the characteristics of the annotations used to represent variability. Some techniques allow to define the mapping between modeling artifacts and products by explicitly referencing features or feature expres-

sions. Other techniques allow to express notions of variability such as optionality or alternativity to represent the variability of a product line without including information about the mapping in terms of features, i.e., they do not establish a mapping to an explicit variability model. A potential limitation of existing annotation-based modeling techniques is that they are mainly intented to be used as a foundational representation for analyses rather than as a tool to be used by practicioners. A potential problem is that a single model for a product line may easily become to large to be handled efficiently, especially in the presence of variability annotations. For future research, we suggest to investigate this assumption and potential concepts to improve the usability of annotation-based approaches, e.g., by introducing concepts for decomposition of a large model into smaller parts.

The commonality of composition-based techniques is that the behavior of individual features is encapsulated into separate modules that may be composed to assemble desired products. Individual approaches mainly differ in the characteristics of the composition mechanism. Again, there are approaches that are mainly intended as underlying formalism for verification based on TS or I/O-automata. We have identified the expressiveness of feature modules as a potential limitation in these approaches. For instance, in many approaches features can perform changes only to predefined locations in the base system. We raise the question whether such mechanisms of feature modularization are expressive enough to express a notion of feature refinement as known from implementation techniques. For composition-based modeling, there also exist approaches based on higher-level modeling languages that are intended to be more suitable as a means for practicioners to model product lines. A simple approach is to use existing modeling techniques for single systems and map parts of a model to features that can be aggregated to generate products. This approach has been shown to be sufficient in some cases, but it is limited to one dimension of decomposition, i.e., they cannot be used to modularize crosscutting features. To overcome this limitation, composition mechanisms known from implementation techniques for product lines have been applied to modeling languages. We propose to bridge the gap between these higher-level composition-based modeling languages and the formalims based on transition systems to benefit from both the usability and possibility of efficient analysis.

4. CONCLUSION

We have presented a survey of formal modeling techniques for product lines to give an overview of existing approaches that can serve as an introduction for researchers who want to develop new techniques. Our results show, that many existing approaches share commonalities regarding the handling of variability. We distinguish between annotation-based and composition-based techniques and exemplify them by means of a running example to illustrate their differences. The current scientific debate primarily centers around verification mechanisms rather than modeling techniques. We see our work as a first step towards a unifying view on variability mechanisms in formal modeling techniques and eventually for all kind of software artifacts. We think that a unified view also helps to understand the differences of individual approaches. In future work, the survey should be extended by comparing in more detail the expressiveness, the complexities, the requirements of tool support, and the performance of different modeling techniques.

5. ACKNOWLEDGMENTS

This work has been partially funded by the German Federal Ministry of Education and Research (01IS14017A, 01IS14017B).

References

[1] M. Alférez, A. Moreira, and J. Araújo. Evaluating Scenario-Based SPL Requirements Approaches — The Case for Modularity, Stability and Expressiveness. *Requirements Engineering*, pages 1–22, 10 2013.

[2] K. Altisen, F. Maraninchi, and D. Stauch. Aspect-oriented Programming for Reactive Systems: Larissa, a Proposal in the Synchronous Framework. *Science of Computer Programming (SCP)*, 63(3):297–320, Dec. 2006.

[3] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, Berlin, Heidelberg, 2013.

[4] S. Apel, W. Scholz, C. Lengauer, and C. Kästner. Detecting Dependences and Interactions in Feature-Oriented Design. In *Proc. Int'l Symposium Software Reliability Engineering (ISSRE)*, pages 161–170, Washington, DC, USA, 2010. IEEE.

[5] P. Asirelli, M. H. ter Beek, A. Fantechi, and S. Gnesi. A Compositional Framework to Derive Product Line Behavioural Descriptions. In *Proc. Int'l Symposium Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 146–161, Berlin, Heidelberg, Oct. 2012. Springer.

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

[7] D. Batory and E. Börger. Modularizing Theorems for Software Product Lines: The Jbook Case Study. *J. Universal Computer Science (J.UCS)*, 14(12):2059–2082, 2008.

[8] D. Beuche. *Composition and Construction of Embedded Software Families*. PhD thesis, University of Magdeburg, Germany, 2003.

[9] E. Börger and R. F. Stark. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, Secaucus, NJ, USA, 2003.

[10] M. Calder and A. Miller. Feature Interaction Detection by Pairwise Analysis of LTL Properties—A Case Study. *Formal Methods in System Design*, 28(3):213–261, 2006.

[11] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, Massachussetts, 1999.

[12] E. M. Clarke and J. M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, Dec. 1996.

[13] A. Classen, M. Cordy, P. Heymans, A. Legay, and P.-Y. Schobbens. Model Checking Software Product Lines with SNIP. *Int'l J. Software Tools for Technology Transfer (STTT)*, 14(5):589–612, 2012.

[14] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Trans. Software Engineering (TSE)*, 39(8):1069–1089, Aug. 2013.

[15] 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'l Conf. Software Engineering (ICSE)*, pages 335–344, New York, NY, USA, 2010. ACM.

[16] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, USA, 2001.

[17] M. Cordy, A. Classen, P. Heymans, A. Legay, and P.-Y. Schobbens. Model Checking Adaptive Software with Featured Transition Systems. In *Proc. Workshop Assurances for Self-Adaptive Systems (ASAS)*, pages 1–29, Berlin, Heidelberg, 2013. Springer.

[18] M. Cordy, A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. ProVeLines: A Product Line of Verifiers for Software Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 141–146, New York, NY, USA, 2013. ACM.

[19] M. Cordy, A. Classen, G. Perrouin, P.-Y. Schobbens, P. Heymans, and A. Legay. Simulation-Based Abstractions for Software Product-Line Model Checking. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 672–682, Piscataway, NJ, USA, 2012. IEEE.

[20] M. Cordy, P.-Y. Schobbens, P. Heymans, and A. Legay. Behavioural Modelling and Verification of Real-Time Software Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 66–75, New York, NY, USA, 2012. ACM.

[21] M. Cordy, P.-Y. Schobbens, P. Heymans, and A. Legay. Towards an Incremental Automata-Based Approach for Software Product-Line Model Checking. In *Proc. Int'l Workshop Formal Methods and Analysis in Software Product Line Engineering (FMSPLE)*, pages 74–81, New York, NY, USA, 2012. ACM.

[22] M. Cordy, P.-Y. Schobbens, P. Heymans, and A. Legay. Beyond Boolean Product-Line Model Checking: Dealing with Feature Attributes and Multi-Features. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 472–481, Piscataway, NJ, USA, May 2013. IEEE.

[23] X. Devroey, M. Cordy, G. Perrouin, E.-Y. Kang, P.-Y. Schobbens, P. Heymans, A. Legay, and B. Baudry. A Vision for Behavioural Model-Driven Validation of Software Product Lines. In *Proc. Int'l Symposium Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 208–222, Berlin, Heidelberg, 2012. Springer.

[24] A. Fantechi and S. Gnesi. Formal Modeling for Product Families Engineering. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 193–202, Washington, DC, USA, 2008. IEEE.

[25] D. Fischbein, S. Uchitel, and V. Braberman. A Foundation for Behavioural Conformance in Software Product Line Architectures. In *Proc. Int'l Workshop Role of Software Architecture for Testing and Analysis (ROSATEA)*, pages 39–48, New York, NY, USA, 2006. ACM.

[26] K. Fisler and S. Krishnamurthi. Modular Verification of Collaboration-Based Software Designs. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 152–163, New York, NY, USA, 2001. ACM.

[27] H. Gomaa and M. Hussein. Dynamic Software Reconfiguration in Software Product Families. In F. van der Linden, editor, *PFE*, volume 3014 of *Lecture Notes in Computer Science*, pages 435–444. Springer, 2003.

[28] A. Gondal, M. Poppleton, and M. Butler. Composing Event-B Specifications: Case-Study Experience. In *Proc. Int'l Symposium Software Composition (SC)*, pages 100–115, Berlin, Heidelberg, 2011. Springer.

[29] J. Greenyer, A. M. Sharifloo, M. Cordy, and P. Heymans. Efficient Consistency Checking of Scenario-Based Product-Line Specifications. In *Proc. Int'l Conf. Requirements Engineering (RE)*, pages 161–170, Piscataway, NJ, USA, Sept. 2012. IEEE.

[30] A. Gruler, M. Leucker, and K. Scheidemann. Modeling and Model Checking Software Product Lines. In *Proc. IFIP Int'l Conf. Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 113–131, Berlin, Heidelberg, 2008. Springer.

[31] M. Janota, J. Kiniry, and G. Botterweck. Formal Methods in Software Product Lines: Concepts, Survey, and Guidelines. Technical Report Lero-TR-SPL-2008-02, Lero, University of Limerick, May 2008.

[32] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *Proc. Int'l Symposium Formal Methods for Components and Objects (FMCO)*, pages 142–164, Berlin, Heidelberg, 2012. Springer.

[33] 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 CMU/SEI-90-TR-21, Software Engineering Institute, 1990.

[34] S. Katz. Aspect Categories and Classes of Temporal Properties. *Trans. Aspect-Oriented Software Development*, 1:106–134, 2006.

[35] R. M. Keller. Formal Verification of Parallel Programs. *Comm. ACM*, 19(7):371–384, July 1976.

[36] K. G. Larsen, U. Nyman, and A. Wąsowski. Modal I/O Automata for Interface and Product Line Theories. In *Proc. Europ. Conf. Programming (ESOP)*, pages 64–79, Berlin, Heidelberg, 2007. Springer.

[37] K. Lauenroth, A. Metzger, and K. Pohl. Quality Assurance in the Presence of Variability. In *Intentional Perspectives on Information Systems Engineering*, pages 319–333, Berlin, Heidelberg, 2010. Springer.

[38] K. Lauenroth, K. Pohl, and S. Toehning. Model Checking of Domain Artifacts in Product Line Engineering. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 269–280, Washington, DC, USA, 2009. IEEE.

[39] H. Li, S. Krishnamurthi, and K. Fisler. Interfaces for Modular Feature Verification. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 195–204, Washington, DC, USA, 2002. IEEE.

[40] H. Li, S. Krishnamurthi, and K. Fisler. Verifying Cross-Cutting Features as Open Systems. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, pages 89–98, New York, NY, USA, Nov. 2002. ACM.

[41] H. Li, S. Krishnamurthi, and K. Fisler. Modular Verification of Open Features Using Three-Valued Model Checking. *Automated Software Engineering*, 12(3):349–382, July 2005.

[42] J. Liu, S. Basu, and R. Lutz. Compositional Model Checking of Software Product Lines Using Variation Point Obligations. *Automated Software Engineering*, 18(1):39–76, 2011.

[43] M. Lochau, S. Mennicke, H. Baller, and L. Ribbeck. DeltaCCS: A Core Calculus for Behavioral Change. In T. Margaria and B. Steffen, editors, *Proc. Int'l Symposium Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 320–335, Berlin, Heidelberg, Oct. 2014. Springer.

[44] J.-V. Millo, S. Ramesh, S. N. Krishna, and G. K. Narwane. Compositional Verification of Software Product Lines. In *Proc. World Conf. Integrated Formal Methods (iFM)*, pages 109–123, Berlin, Heidelberg, 2013. Springer.

[45] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, Apr. 1989.

[46] R. Muschevici, D. Clarke, and J. Proenca. Feature Petri Nets. In *Proc. Int'l Workshop Formal Methods and Analysis in Software Product Line Engineering (FMSPLE)*, pages 99–106, Lancaster, UK, Sept. 2010. Lancaster University.

[47] T. Nelson, D. D. Cowan, and P. S. C. Alencar. Supporting Formal Verification of Crosscutting Concerns. In *Proc. Int'l Conf. Metalevel Architectures and Separation of Crosscutting Concerns*, pages 153–169, London, UK, 2001. Springer.

[48] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, Berlin, Heidelberg, Sept. 2005.

[49] M. Poppleton. Towards Feature-Oriented Specification and Development with Event-B. In *Proc. Int'l Working Conf. Requirements Engineering: Foundation for Software Quality (REFSQ)*, pages 367–381, Berlin, Heidelberg, 2007. Springer.

[50] G. Püschel, R. Seiger, and T. Schlegel. Test Modeling for Context-aware Ubiquitous Applications with Feature Petri Nets. In *Proc. Workshop Model-based Interactive Ubiquitous Systems (MODIQUITOUS)*, 2012.

[51] H. Sabouri and R. Khosravi. Efficient Verification of Evolving Software Product Lines. In *Proc. Int'l Conf. Fundamentals of Software Engineering (FSEN)*, pages 351–358, Berlin, Heidelberg, 2012. Springer.

[52] H. Sabouri and R. Khosravi. Delta Modeling and Model Checking of Product Families. In *Proc. Int'l Conf. Fundamentals of Software Engineering (FSEN)*, pages 51–65, Berlin, Heidelberg, 2013. Springer.

[53] I. Schaefer, D. Gurov, and S. Soleimanifard. Compositional Algorithmic Verification of Software Product Lines. In *Proc. Int'l Symposium Formal Methods for Components and Objects (FMCO)*, pages 184–203, Berlin, Heidelberg, Nov. 2010. Springer.

[54] I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, and K. Villela. Software Diversity: State of the Art and Perspectives. *Int'l J. Software Tools for Technology Transfer (STTT)*, 14:477–495, 2012.

[55] H. Sipma. A Formal Model for Cross-Cutting Modular Transition Systems. In *Proc. Workshop Foundations of Aspect-Oriented Languages (FOAL)*, 2003.

[56] J. Sorge, M. Poppleton, and M. Butler. A Basis for Feature-Oriented Modelling in Event-B. In *Proc. Int'l Conf. Abstract State Machines, Alloy, B and Z (ABZ)*, pages 409–409, Berlin, Heidelberg, 2010. Springer.

[57] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 107–119, New York, NY, USA, 1999. ACM.

[58] M. H. ter Beek, A. L. Lafuente, and M. Petrocchi. Combining Declarative and Procedural Views in the Specification and Analysis of Product Families. In *Proc. Int'l Workshop Formal Methods and Analysis in Software Product Line Engineering (FMSPLE)*, pages 10–17, New York, NY, USA, 2013. ACM.

[59] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys*, 47(1):6:1–6:45, June 2014.

[60] M. Tribastone. Behavioral Relations in a Process Algebra for Variants. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 82–91, New York, NY, USA, 2014. ACM.

[61] D. M. Weiss. The Product Line Hall of Fame. In *Proc. Int'l Software Product Line Conf. (SPLC)*, page 395, Washington, DC, USA, 2008. IEEE.