Towards Reasoning about Product Lines with Design Choices

Navpreet Kaur
Université de Montréal
kaurnavp@iro.umontreal.ca

Michalis Famelis
Université de Montréal
famelis@iro.umontreal.ca

***Abstract*—While designing changes to Software Product Lines (SPLs), engineers may need to consider many alternative SPL designs. In the absence of enough information to pick an appropriate SPL design, they face design-time uncertainty about how to make the appropriate design choices. The combination of the two dimensions (variability and design choices) leads to Software Product Lines with Design Choices (SPLDCs). We propose Tyson, an Alloy-based domain-specific language for modelling SPLDCs and reasoning about their structural properties. We illustrate the applicability and feasibility of Tyson with a worked example, showing the kind of nuanced feedback necessary for meaningful analysis of SPLs with design choices.**

I. Introduction

Software Product Line (SPL) engineering allows integrating mass customization of software products by integrating multiple software variants in a common platform [24]. SPLs allow organizations to make long-term commitments to the maintenance of variability in families of related products by modelling their commonalities and differences. Variability is typically modelled using features and their inter-dependencies, typically in a *feature model* [26], [24].

Consider the toy example of a company that develops the software controller for an automated washing system. The company has clients with different needs and has therefore developed a family of products i.e., a family of software controllers called WM. Here we assume that the company uses models to represent its various software artifact [5]. We show the WM SPL in Fig. 1; it contains a feature model and a base *domain model*. The domain model of WM is a (simplified) UML State Machine, showing all the possible states and transitions for washing machine variants. The feature model has four features: *Wash*, *Heat*, *Delay* and *Dry* that can be combined to generate several variants. *Wash* is a mandatory feature, i.e., it must be present in all the valid feature configurations; the other three features are optional. To generate a variant, the developers must select a valid subset of its features. The variant is then generated by evaluating the *presence conditions* of the elements of the domain model, shown in Fig. 1(b) using grey boxes. E.g., selecting features *Wash*, *Heat* and *Delay* generates the washing machine variant shown in Fig. 2. In this case, the washing machine initially locks the door, and then enters a `Waiting` state for a predetermined period of time. Once that time is over, it enters the `Washing` state, doing a `TempCheck` at the start; once that is done, the door is unlocked and the process terminates.

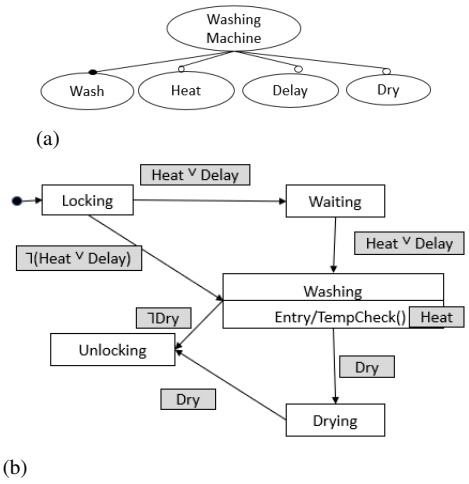


Fig. 1. WM product line: (a) Feature Model, (b) Domain Model

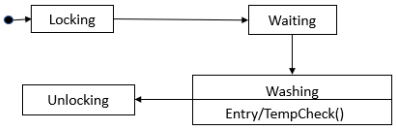


Fig. 2. A variant of WM with features *Heat* and *Delay*

While SPL engineering allows the long-term maintenance of variability, engineers working on it may need to express and reason about *short-term* design choices. These choices are a source of *design uncertainty* [25] and can result from scenarios such as dealing with different design alternatives, making decisions about product architecture, resolving model inconsistencies, or resolving conflicting stakeholder requirements. E.g., the developers may not know whether the *Heat* feature should be explicitly modelled as happening incrementally, by adding a self-loop on the `Waiting` state of the domain model. However, contrary to variability, design choices can affect all elements of an SPL definition. E.g., developers may be unsure if the features *Heat* and *Delay* should be made mutually exclusive or not. We represent such uncertain design decisions as boolean choices. In Fig. 3, these two design decisions are shown as yellow annotation boxes with dashed outlines on the SPL model elements that are impacted by the decision. Specifically, *Mutex* is shown as an annotation to the mutual exclusion arc between the Heat and Delay features in Fig. 3(a), and *IncrementalHeat* as an annotation to a self-transition on

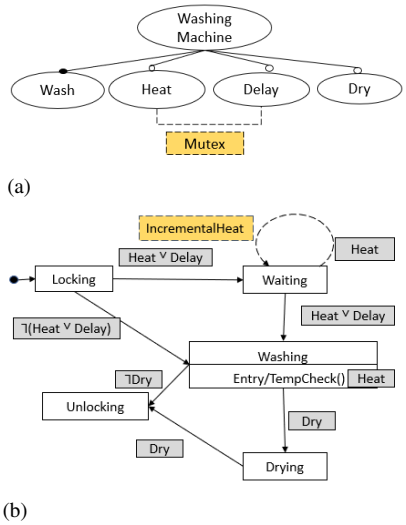


Fig. 3. WM SPL with the design choices *Mutex* and *IncrementalHeat* : (a) Feature Model, (b) Domain Model.

the state Waiting, as shown in Fig. 3(b). Different answers to these design decisions lead developers to different SPL designs. For example, if they decide against both *Mutex* and *IncrementalHeat*, the resulting SPL is the one shown in Fig. 1.

We call the systems that exhibit both variability and uncertainty about design decision *Software Product Lines with Design Choices* (SPLDCs) [10]. In previous work, we showed that instead of finalizing design decisions with little knowledge about their impact, developers can defer decision making until they acquire more information [9]. However, that work focused on single products rather than product families. In SPLDCs we get a combination of both long term variability commitments and short term design choices that cannot be adequately addressed by the state of the art.

Suppose the developers need to check the satisfaction of system properties, such as consistency while deferring the resolution of design decisions. E.g., consider the property R_2 in Table I. Developers might want to make sure that whatever decision they make in the future, and whatever configuration a client might choose, the state machine of the resulting controller product will always contain a final state. Answering such questions is a non trivial task, as both variability and design uncertainty must be taken into account. The variability of features and the uncertainty among design choices gives a set of software product lines, each of which itself is a set of different products having some commonality. In other words, the analysis must be performed over a powerset of products. While many approaches exist to manage and reason about variability in product lines [28], [3], and some techniques are developed for uncertainty management [9], there is no significant work to reason in the presence of both dimensions. Furthermore, existing work that considers two-dimensional SPLs, e.g., across time [27], or that combines feature and decision modelling [6], do not allow formal reasoning as described above.

In this paper we focus on reasoning about properties of

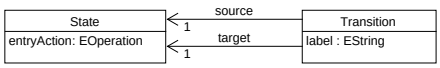


Fig. 4. Simplified state machine metamodel

SPLDCs, a problem first posited in our previous vision paper [10]. We provide a decision procedure and introduce Tyson, a modelling language with Alloy-based semantics [13]. We outline a reasoning approach to checking properties of Tyson models that allows generating feedback that informs users of the causes of a failed property check by separating the two dimensions. We discuss background in Sec. II and introduce the SPLDC formalism in Sec. III. In Sec. IV, we outline SPLDC property checking. In Sec. V we discuss Tyson's implementation and evaluate it in Sec. VI. We discuss related work in Sec. VII and conclude in Sec. VIII.

II. Background

We assume the *annotative* SPL paradigm [15]. A more detailed treatment can be found in [16]. A **Software Product Line** (SPL) S is a 3-tuple $< FM, DM, \mu >$ made up of a Feature Model *FM*, a Domain Model *DM*, and feature mapping μ that maps features to the domain model entities. E.g., Fig. 1 represents the SPL of a washing machine controller. A **feature model** FM is a graphical representation of a 2-tuple, $< F, \Phi_F >$ where F is a set of features, and Φ_F is a formula representing variability constraints among them. The feature model of WM is shown in Fig. 1(a) with features *Wash*, *Heat*, *Delay*, and *Dry*, and the constraint that *Wash* is mandatory, i.e., it must be present in all valid configurations. A **domain model** DM is a graphical representation of a set of various model elements *D*, and a formula ϕ_D that represents the metamodel and well-formedness constraints. The domain model of WM is the simplified UML State Machine, shown in Fig. 1(b). Its well-formedness constraints are expressed in the simplified metamodel shown in Fig. 4. A **feature mapping** μ is a function $\mu : FM \rightarrow DM$, consisting of a set of tuples $\langle E, \phi_E \rangle$ that map each entity E of the domain model with a propositional formula ϕ_E over the features from the FM, known as a **presence condition**. We represent presence conditions graphically using grey box annotations next to the graphical elements that they apply to. E.g., in WM the state *Drying* has the presence condition ϕ_{Drying} =*Dry*.

A **feature configuration** ρ of a feature model $FM =< F, \Phi_F >$ is a subset of features from F that satisfies Φ_F. The set of all feature configurations of a feature model FM is denoted by $Conf(FM)$. For example, for WM, some of the feature configurations are: $\{Wash, Dry\}$, $\{Wash, Heat\}$, $\{Wash, Delay, Dry\}$ Given a valid feature configuration ρ, a product M is **derived** from an SPL, such that only those elements are present in its domain model whose presence conditions are satisfied under ρ. The set of all products that can be derived by a product line SPL is denoted by $Conf(SPL)$ For example, the WM variant represented in Fig. 2, is a

TABLE I
PRODUCT-LEVEL PROPERTIES FOR THE WM EXAMPLE

#	SPLDC	Property
R_1	WM	∃ state S in DM: s is an initial State.
R_2	WM	∃ state S in DM: s is a final State.
R_3	WM	∀ transition T in DM: T has a guard

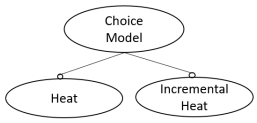


Fig. 5. Choice Model for WM SPLDC

product derived from SPL using the feature configuration: $\rho = \{Wash, Heat, Delay\}$.

As introduced in [10], a property R that constrains a model without any variability is called a **product level property**. In other words, a product level property concerns individual products, rather than entire SPLs. Here, we focus on structural properties, which we assume expressed in first-order logic. A set of product-level properties for products of WM are shown in Table I. E.g., in a WM product the property R_2 "the model has a final state" can be expressed as[1] $\exists s : State \nexists t : Transition \cdot (t.source = s)$

III. MODELLING DESIGN CHOICES IN SPLS

Design uncertainty can affect any part of a product line definition for which modellers need to make a design decision. This includes uncertainty in the design of the domain model (like the choice *IncrementalHeat* in WM), of the feature model (like the choice *Mutex* in WM) or of the feature mapping.

A **design choice** is a propositional variable that encodes the choice of a particular solution to a design problem. So as long as a design choice variable is not bound to True or False, it represents the modeller's uncertainty about that choice. For example, the design choice *Mutex* is a propositional variable encoding the uncertainty that modellers have whether the features *Heat* and *Delay* should be mutually exclusive. A **choice model** CM, is a graphical representation of a tuple $< C, \phi_C >$, where C is a set of design choices C, and ϕ_C a formula representing dependencies between them. For example, the choice model for WM, is shown in Fig. 5, using the concrete syntax of feature models. It consists of the two optional decisions *Mutex* and *IncrementalHeat*. The design choices that capture the design uncertainty of modellers about an SPL, can then be mapped to the SPL elements. A **decision mapping** δ is a function $\delta : CM \rightarrow SPL$ consisting of a set of tuples $< S, \phi_S >$, mapping each entity S of an SPL to a propositional formula ϕ_S defined over the SPL entities with respect to the choices in CM. For instance, in WM (Fig. 3(b)) the transition looping over state *Waiting* is present in a product iff the choice *IncrementalHeat* is selected.

This preliminary approach to modelling uncertainty about design choices follows very closely the feature modelling approach. This is because, following previous work on partial models [11], we make two key assumptions: (a) that modellers are aware of all the design choices about the SPL that need to be addressed in the short term, and (b) that for each such choice, a set of possible acceptable solutions has been elicited. This allows us to represent design choices as boolean choice variables. We are currently working on extending the partial modelling approach to relax our two assumptions and allow more sophisticated modelling of design uncertainty, taking into account existing work on modelling decisions in SPLs [6]. Regardless, as discussed in Sec. I, here we focus on the feasibility of reasoning about properties of SPLDCs and providing nuanced feedback. This simple modelling approach still allows to explore the topic without loss of generality.

A **Software Product Line with Design Choices** (SPLDC) is a tuple $\langle CM, SPL, \delta \rangle$ where *SPL* is a product line definition, CM is a choice model and δ is the decision mapping between them. For instance, Fig. 3 represents an SPLDC of WM controller, where yellow boxes represent the design choices *Mutex* and *IncrementalHeat*.

Given the choice model $CM =< C, \phi_C >$ of an SPLDC SC, a valid **design decision** α is a subset of design choice variables from C that satisfy ϕ_C. In other words, if in ϕ_C we substitute all variables in α by *true* and all other variables by *false*, the resulting expression evaluates to *true*. The set of all valid design decisions of SC is denoted as $Ch(SC)$. In the WM example we have that: Ch(WM) ={ { }, {*Mutex*, *IncrementalHeat*}, {*Mutex*}, {*IncrementalHeat*}}. A **concretization** is a model without design uncertainty that results from resolving all design uncertainty in a partial model [11]. Here, a concretization n of an SPLDC $S = \langle CM, S, \delta \rangle$ is an SPL that can be derived from S under a design decision α, such that it contains only elements of S that are mapped by δ under α. The set of all concretizations that can be derived from S is denoted by $Ch(S)$. For example the SPL shown in Fig. 1 is a concretization of the WM SPLDC that can be derived using the decision $\alpha = \{\}$ (i.e., all choice variables false).

IV. REASONING ABOUT SPLDCS

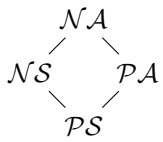
In previous work [10], we identified four useful levels of satisfaction of a product-level property R and an SPLDC SC: $\mathcal{NA}$, $\mathcal{NS}$, $\mathcal{PA}$, and $\mathcal{PS}$, shown in Table II. We explain the intuition of these levels for the property R_2 from Table IV. For an SPLDC like WM, the levels of satisfaction of R_2 have the following meaning: $\mathcal{NA}$: regardless what decisions are made to create an SPL, every possible configuration will lead to a state machine with final state. $\mathcal{NS}$: there is a set of design decisions that would lead to an SPL design for which every possible configuration will lead to a state machine with final state. $\mathcal{PA}$: regardless what decisions are made, it is always possible to configure the resulting SPL such that a state machine can be derived with final state. $\mathcal{PS}$: there is a set of design decisions that would lead to an SPL design which is possible to configure to derive a state machine with final state.

[1] To maintain the simplicity of the metamodel of our toy example, we assume that a final state is on that has no outgoing transition.

TABLE II
LEVELS OF SATISFACTION OF PRODUCT-LEVEL PROPERTY R FOR SPLDC K (ADAPTED FROM PREVIOUS WORK [10]).

	All Products		Some Products	
	Necessary for product line	**Possible for product line**	**Necessary for product line**	**Possible for product line**
Level	$\mathcal{NA}$	$\mathcal{PA}$	$\mathcal{NS}$	$\mathcal{PS}$
Property R holds in...	every product of every product line	every product of at least one product line	at least one product of every product line	at least one product of at least one product line
Formalization $\mathcal{F}$	$\forall\alpha \cdot \Phi_K^{\exists} \Rightarrow (\forall\rho \cdot \Phi_K \Rightarrow R)$	$\exists\alpha \cdot \Phi_K^{\exists} \wedge (\forall\rho \cdot \Phi_K \Rightarrow R)$	$\forall\alpha \cdot \Phi_K^{\exists} \Rightarrow (\exists\rho \cdot \Phi_K \wedge R)$	$\exists\alpha \cdot \Phi_K^{\exists} \wedge (\exists\rho \cdot \Phi_K \wedge R)$
Counterexample	α, ρ	ρ	α	–

Using logical consequence as a binary relation between levels, the four levels form a partially ordered set (*poset*), where $\mathcal{PS}$ is the minimal and $\mathcal{NA}$ is the maximal element, and where $\mathcal{NS}$ and $\mathcal{PA}$ are at same rank. Given the lemma of Bjorner [2] that a bounded poset of finite rank forms a lattice, the four SPLDC satisfaction levels form the lattice $\mathcal{L}$ shown on the right.



We use this to define an analysis procedure, that given as inputs an SPLDC K and a product-level property R, produces as output he level of satisfaction of R in K.

First, we encode the SPLDC K in logic, by constructing the formula: $\Phi_K = \Phi_D \wedge \Phi_F \wedge \bigwedge_{e \in E} \phi_e \wedge \Phi_C \wedge \bigwedge_{s \in S} \phi_s$, where, Φ_D encodes the domain model, Φ_F the feature model, the formulas ϕ_e over the set E of Domain Model elements the feature mapping, Φ_C the choice model, and the set of formulas ϕ_s over the set S of elements of the SPL (features, domain model elements and feature mapping tuples) the decision mapping. The formula Φ encodes the entire two dimensional space of SPLs and products; a valid design decision α and a valid configuration ρ define exactly one satisfying assignment.

To check if a property is satisfied at a particular level, we check the validity of the corresponding logical formalization $\mathcal{F}$ of that level, shown on the third row of Table II, using a SAT solver such as Sat4J [18]. The formulas $\mathcal{F}_{\mathcal{NA}}$,$\mathcal{F}_{\mathcal{PA}}$,$\mathcal{F}_{\mathcal{NS}}$, and $\mathcal{F}_{\mathcal{PS}}$ *lift* product-level properties to the SPLDC level, allowing the quantification over design choices and features. The formula Φ_K is satisfied for combinations of decision and feature variables. However, when reasoning about SPLDCs, we want to be able to provide nuanced feedback to users, separating the cause of analysis results into each dimension. We therefore use the formula $\Phi_K^{\exists}$ which encodes the dependency constraints on just the design choice variables; it is derived from Φ_K by *quantifying out* [31] all variables except for those representing design choices. Each combination of values assigned to the design choice variables is a design decision α, i.e., a set of decisions in the design space that define a single SPL design. Each subsequent combination of feature variables is a product configuration ρ of that SPL design.

In case a property is not satisfied at a particular level, we produce appropriate feedback by using the counterexample generated by the SAT solver during the validity check. This consists of a truth assignment of the design choice variables in α and/or the feature variables in ρ depending on the level. We translate this truth assignment back to the level of abstraction used for SPLDC modelling and present it to the user as a counterexample. There is an obvious tradeoff between generating nuanced feedback that separates between the two dimensions (variability and design choices). The computation requires the existential quantification over the two different sets of variables and the check for satisfiability, and is therefore computationally costly.

To analyse the overall level of satisfaction of a requirement R, we start our analysis at the maximal element of $\mathcal{L}$, and move downwards. At rank 2, the order of checking $\mathcal{F}_{\mathcal{PA}}$ and $\mathcal{F}_{\mathcal{NS}}$ is irrelevant. In the worst case (where the property is not satisfied by any product of any possible product line design), four checks are required in total.

Suppose we want to analyze requirement property R_1 ("there is an initial state") given in Table I, that is expressed by the formula $R_1 = \exists s : State \nexists t : Transition \cdot (t.target = s)$. First we check whether its level of satisfaction is $\mathcal{NA}$. To do this, we construct $\mathcal{F}_{\mathcal{NA}}$ using the logical encoding of the WM SPLDC and R_1, and check whether it is valid, using a SAT solver. We find that it is, and therefore we neither need to check the other levels, nor to produce any further feedback to the modellers.

Suppose that we then want to analyze the property R_3 ("there exists a transition that has a guard") from Table I. Following the same process, we find that it is not satisfied for $\mathcal{NA}$. To help modellers understand why, we pinpoint a product that violates the property, by providing a set of design and configuration choices. The exact choice of feedback exemplar depends on the SAT solver, but one possibility is α={*Mutex*} and ρ={*Wash*,*Dry*}. Going down the lattice $\mathcal{L}$, we check whether R_3 is satisfied for level $\mathcal{NS}$, also getting a counterexample. In this case, the counterexample is a design decision α that results in an SPL concretization with no products that satisfy the property. A possible feedback generated by the solver is α={*Mutex*,*IncrementalHeat*} as resulting the SPL has no product that contains a transition with a guard. We also check whether R_3 is satisfied for level $\mathcal{PA}$, which also generates as a counterexample a configuration that exists in every product line and for which the property is not satisfied. A possible feedback generated by the solver is ρ={*Wash*}: the simplest washing machine configuration is included in all SPL concretizations and does not contain any transitions with guards. Finally, we check the lowest level of $\mathcal{L}$, i.e., whether R_3 is satisfied at level $\mathcal{PS}$. We find that it

is not (as the domain model, shown in Fig. 3(b), does not contain any guards) and the solver does not need to generate a counterexample, as by definition every combination of α and ρ is a counterexample.

Theorem: Given an SPLDC K, encoded logically as Φ_K, and a product-level property R, the level of satisfaction of R is $\mathcal{NA}$ iff the formula $\mathcal{F}_{\mathcal{NA}}$ (R)= $\forall\alpha \cdot \Phi_K^{\exists} \Rightarrow (\forall\rho \cdot \Phi_K \Rightarrow R)$ is valid.

Proof: The satisfaction level of R in K is $\mathcal{NA}$ if $\forall s \in Ch(K), \forall p \in Conf(s) \cdot p \models R$. Assume that there is an SPL concretization s in $Ch(K)$ from which we can derive a product p in $Conf(s)$ such that $p \not\models R$. That means that there is a design decision α_s that can be used to derive s from K and that there is a configuration ρ_p that can be used to derive p from s. But since the formula $\mathcal{F}_{\mathcal{NA}}$ (R) is valid, it is true for every design decision α and configuration ρ. Therefore there is no way to derive a p such that $p \not\models R$ and thus the satisfaction level of R for K is $\mathcal{NA}$. Conversely, if the level of R in K is $\mathcal{NA}$, then for every s in $Ch(K)$, i.e., for every decision α_s, every product p in $Conf(s)$, i.e., for every configuration ρ_p, it is the case that $p \models R$. Therefore $\mathcal{F}_{\mathcal{NA}}$ (R) is valid. □

The theorems and proofs for $\mathcal{PA}$, $\mathcal{NS}$, and $\mathcal{PS}$ are analogous.

Listing 1. Tyson model of the WM example

```
1  CM { Mutex; IncrementalHeat; }
2
3  FM { Wash; Delay; Dry; Heat;
4     FMConst: [ Mandatory (Wash) ] }
5
6  StateChart {
7    State Locking; State Waiting;
8    State Washing; State Drying;
9    State UnLocking;
10   Transition T1: Locking to Waiting
11   Transition T2: Waiting to Washing
12   Transition T3: Locking to Washing
13   Transition T4: Washing to UnLocking
14   Transition T5: Washing to Drying
15   Transition T6: Drying to UnLocking
16   Transition T7: Waiting to Waiting
17 }
18
19 Mappings {
20   FMap {
21     F1: {(Wash IN ) =>
22       AND (Transition T3 IN, Transition T4 IN)}
23     F2: {OR (Heat IN, Delay IN) =>
24       AND (Transition T1 IN, Transition T2 IN)}
25     F3: {(Dry IN) =>
26       AND (Transition T5 IN, Transition T6 IN)}
27   }
28   DMap {
29     D1: {Mutex IN =>
30       XOR (Feature Heat IN, Feature Delay IN)}
31     D2: {IncrementalHeat IN => Transition T7 IN}
32   }
33 }
```

V. Implementation

Using Xtext [8], we implemented *Tyson*, a language for modelling SPLDCs and checking their properties. Our focus is reasoning so in this version of Tyson we made some simplifications regarding modelling: SPLDC models are monolithic, and are expressed in a purely textual concrete syntax. Tyson currently supports simplified UML class diagrams and state machines as domain models, allowing us to experiment with both behavioural and structural models. In the future, we intend to adopt a more modular modelling approach, which would allow expressing SPLDCs for arbitrary modelling languages and concrete syntaxes. We envision reusing the current syntax as an intermediate representation. Below, we illustrate Tyson using the WM example. The implementation is accessible at https://bitbucket.org/Navpreet15/dsl/src/master/.

A. Tyson's Syntax

The Tyson model is shown in Listing 1 and can be accessed in full in [16]. The Tyson model has 4 parts, corresponding to the components of the WM SPLDC: (a) the domain model (lines 6-17), (b) the choice model (line 1), (c) the feature model (lines 3-4), and (d) the feature and decision mappings (lines 19-33). We outline these below.

The domain model corresponds to the State Machine in Fig. 3(b) and has 5 states (lines 7-9) and 7 Transitions (T1-T7, in lines 10-16). The feature model shown in Fig. 3(a) having 4 features is specified in lines 3-4. We also define a constraint that *Wash* is a Mandatory feature. Tyson also allows the definition of multiplicity constraints about a feature group. The choice model shown in Fig. 5 with two design choices is specified in line 1, following the same approach (as discussed in Sec. III). Tyson also allows defining constraints about design choices as a part of choice model definition.

The feature mapping is expressed in lines 20-27. The Tyson keyword "IN" signifies the presence of the corresponding element in an SPL or a product. Specifically, *F1*, *F2*, and *F3* are 3 feature mappings, that represent how the presence or absence of a feature effects the entities of domain model. *F1* shows that presence of *Wash* in Feature Model implies that Transition *T2* and *T5* are present in Domain model; otherwise they will be absent. The decision mapping is expressed in lines 28-32, using the same syntax. Specifically, *D1* and *D2* are 2 decision mappings, that represent how the presence or absence of design choices can impact the product line entities. *D1* shows that presence of Design Choice *Mutex* implies that presence of Features *Heat* and *Delay* are mutually exclusive; its absence that they are not.

B. Tyson's Alloy Semantics

In order to reason about the properties of SPLDCs expressed in Tyson, we provide a mapping from Tyson to Alloy, a lightweight formal method that allows bounded reasoning of first order logic [13]. Specifically, Tyson models are transformed into the Alloy specification language using an *Acceleo* model-to-text transformation [23]. Generating optimized Alloy code is future work. Tyson does not provide a property specification language; instead, SPLDC properties are written directly in Alloy. We discuss how properties can be checked for the 4 satisfaction levels in $\mathcal{L}$ in Sec. V-C. After combining the SPLDC with the properties, we use the Alloy Analyzer to translate the overall Alloy specification to a SAT instance using

he Kodkod model finder, which is then checked using a SAT solver as block box reasoner. The full Acceleo translation and full Alloy semantics of the WM example are available in the Appendix of the first author's thesis [16]. Below, we illustrate the translation for each part of the Tyson model in Listing 1.

The translation of the domain model of the Tyson specification (lines 6-17 in Listing 1) into Alloy is shown in Listing 2. The metamodel well-formedness constraint (lines 8-13) states that if a transition is present in a Domain Model, so must its source and target states.

Listing 2. Domain Model Definition in Alloy

```
1  // DOMAIN MODEL DEFINITION
2  // container for model elements, i.e., the domain model
3  sig DomainModel{
4    transition: some Transition,
5    state: some State
6  }
7
8  // Metamodel well-formedness constraints
9  fact{all d: DomainModel | all s : State | some t:
       ↪ Transition |
10   (t in d.transition) and ((s in t.source) or (s in t.
       ↪ target )) ⟹
11   (s in d.state)
12   else (s not in d.state)
13 }
14
15 abstract sig State{}
16 one sig Locking, Waiting, Washing, Drying, Unlocking
       ↪ extends State{}
17
18 abstract sig Transition{
19   source: one State,
20   target: one State
21 }
22
23 one sig T1, T2, T3, T4, T5, T6, T7 extends Transition{}
24 fact {(T1.source = Locking) and (T1.target = Waiting)}
25 fact {(T2.source = Waiting) and (T2.target = Washing) }
26 fact {(T3.source = Locking) and (T3.target = Washing)  }
27 fact {(T4.source = Washing) and (T4.target = Unlocking)
       ↪ }
28 fact {(T5.source = Washing) and (T5.target = Drying)}
29 fact {(T6.source = Drying) and (T6.target = Unlocking) }
30 fact {(T7.source = Waiting) and (T7.target = Waiting)}
```

The translations of the choice and feature models of the Tyson specification (line 1 and lines 3-4 in Listing 1 respectively) are shown in Listing 3 in lines 1-6 and lines 8-14, respectively. The fact in line 14 is the translation in Alloy of the Tyson specification that the feature *Wash* is mandatory.

Listing 3. Choice and Feature Model Definitions in Alloy

```
1  abstract sig Choice{}
2  sig Mutex, IncrementalHeat extends Choice{}
3
4  abstract sig ChoiceModel{
5    choice : set Choice
6  }
7
8  abstract sig Feature{}
9  one sig Wash, Heat , Delay, Dry extends Feature{}
10
11 abstract sig FeatureModel{
12   feature: some Feature
13 }
14 fact {all f: FeatureModel | Wash in f.feature}
```

The translations of the 3 feature mappings of the Tyson specification (lines 19-27 in Listing 1) are shown in Listing 4. Each mapping is expressed as an Alloy *fact*: the first (lines 1-4) corresponds to mapping *F1*, the second (lines 6-9) corresponds to *F2*, and the third (lines 11-14) to *F3*.

Listing 4. Feature Mapping in Alloy

```
1  fact {all p: Product |
2  (Wash in p.config.feature)⟹
3    ((T3 in p.dm.transition) and (T4 in p.dm.transition))
4    else ((T3 not in p.dm.transition) and (T4 not  in p.dm
       ↪ .transition))
5  }
6  fact {all p: Product |
7  (Dry in p.config.feature)⟹
8    ((T5 in p.dm.transition) and (T6 in p.dm.transition))
9    else ((T5 not in p.dm.transition) and (T6 not  in p.dm
       ↪ .transition))
10 }
11 fact {all p: Product|
12 (Heat in p.config.feature) or (Delay in p.config.feature
       ↪ ) ⟹
13   ((T1 in p.dm.transition) and (T2 in p.dm.transition))
14   else ((T1 not in p.dm.transition) and (T2 not  in p.dm
       ↪ .transition))
15 }
```

Based on the domain model, feature model and feature mapping specifications, we can now create an Alloy specification for product lines. This is shown in Listing 5. It defines that a product line contains some products (lines 12-14), where each product contains a valid feature configuration, and a domain model derived by the feature configuration (lines 2-9).

Listing 5. Product Line Definition in Alloy

```
1  //Product Definition
2  abstract sig Product{
3    dm: one DomainModel,
4    config: one FeatureModel
5  }
6
7  //Well formdness rules
8  fact { all f: FeatureModel | f in Product.config }
9  fact { all d: DomainModel  | d in Product.dm     }
10
11 //Product line definition
12 abstract sig SPL{
13   product : some Product
14 }
15 fact { all p: Product | p in SPL.product}
```

The translations of the 2 decision mappings of the Tyson specification (lines 28-32 in Listing 1) are shown in Listing 6. Each one is again expressed as an Alloy fact: the first (lines 2-5) corresponds to the decision mapping *D1* and the second (lines 6-10) to *D2*.

Listing 6. Decision Mapping in Alloy

```
1  //decision Mapping
2  fact { all dec: DesignChoices | all s: dec.spl | all p:
       ↪ s.product |
3    Mutex in dec.cm.choice ⟹
4    not  ((Heat in p.config.feature) and (Delay in p.
       ↪ config.feature))
5  }
6  fact { all dec: DesignChoices | all s: dec.spl | all p:
       ↪ s.product |
7    (IncrementalHeat in dec.cm.choice)  and
8    (Heat in p.config.feature) and (Delay in p.config.
       ↪ feature) ⟹
9    T7 in p.dm.transition
10 }
```

Based on all the above, we can now define the SPLDC specification. This is shown in Listing 7: an SPLDC contains some design choices, where each design choice contains a choice configuration and an SPL derived using the selected choices.

Listing 7. SPLDC definition in Alloy

```
//Design Choices Definition
abstract sig DesignChoices{
  cm  : one ChoiceModel,
  spl : one SPL
}
fact { all c: ChoiceModel | c in DesignChoices.cm }
fact { all s: SPL | s in DesignChoices.spl }

//SPLDC definition
one sig SPLDC{
  dc : some DesignChoices
}
fact { all d: DesignChoices | d in SPLDC.dc }
```

Finally, we define some symmetry breaking constraints, to prevent the Alloy Analyzer from generating structurally equivalent instances. These are shown in Listing 8.

Listing 8. Symmetry Breaking Constraints

```
//symmetry breaking constraints
fact {all f1, f2 : FeatureModel | f1.feature = f2.
    ↪ feature ⟹ f1= f2}

fact {all d1, d2 : DomainModel | d1.transition = d2.
    ↪ transition ⟹ d1=d2}

fact {all t1, t2 : Transition | (t1.source = t2.source)
    ↪ and (t1.target = t2.target) ⟹t1=t2}

fact {all p1, p2 : Product | (p1.config = p2.config) and
    ↪  (p1.dm=p2.dm)⟹p1 = p2}

fact{all c1, c2 : ChoiceModel | c1.choice = c2.choice ⟹
    ↪ c1 = c2}

fact {all dc1, dc2 : DesignChoices | dc1.cm = dc2.cm ⟹
    ↪ dc1=dc2}
```

C. Encoding and Checking Properties

As mentioned before, Tyson does not provide a property specification language; instead, SPLDC properties are written directly in Alloy as assertions. In accordance with the decision procedure outlined in Sec. IV, for a given product-level property we write four assertions, one for each level of satisfaction in $\mathcal{L}$. We show the Alloy encoding of property R_2 from Table I in Listing 9.

Listing 9. SPLDC level properties in Alloy

```
// NA
assert oneFinalNA {all pl: SPL | all p: pl.product| one
    ↪ d: p.dm | one s: d.state |
(s not in p.dm.transition.source)}
// NS
assert oneFinalNS {all pl: SPL | some p: pl.product| one
    ↪  s: p.dm.state |
(s not in p.dm.transition.source) and (s in  p.dm.
    ↪ transition.target)}
// PA
assert oneFinalPA {some pl: SPL | all p: pl.product| one
    ↪  s: p.dm.state |
(s not in p.dm.transition.source) and (s in  p.dm.
    ↪ transition.target)}

// PS
assert oneFinalPS {some pl: SPL | some p: pl.product|
    ↪ one s: p.dm.state |
(s not in p.dm.transition.source) and (s in  p.dm.
    ↪ transition.target)}
```

The Tyson implementation allows declaring product-level properties as raw Strings in the Tyson model, using the Alloy syntax. Tyson then automatically generates SPLDC-level properties according to $\mathcal{L}$. Each property written in Tyson, is therefore transformed to 4 different assertions in Alloy, one for each level in $\mathcal{L}$.

Determining the level of satisfaction of the property, then takes the form of checking the four assertions in the order defined by $\mathcal{L}$, as discussed in Sec. IV. In Alloy, assertion checks are always conducted within a finite bound, in accordance with the small-scope hypothesis [13]. In Tyson, the bound is assumed to be given by the user.

Finally, we rely the Alloy Analyzer's capability to generate counterexamples in order to provide users with nuanced feedback. Specifically, if one of the assertion checks fails, the Alloy Analyzer provides the user with a counterexample, within the given scope (the scope specified for the property checks is a global scope that applies all the signatures), that is entailed by the model but for which the assertion does not hold. By inspecting this counterexample, the user can then understand what design choices and/or configuration options result in the violation.

VI. Evaluation

We aim to evaluate the feasibility and scalability of our approach for analyzing the quality requirements of SPLDCs. We thus pose the following research question: *How is the run time of the various property checks needed for SPLDC analysis affected by the size of the SPLDC?*

A. Setup

We experimented with synthetic but realistic, semi-random SPLDCs of varying sizes, and collected the run times for each check. We checked three consistency properties for each SPLDC and recorded the run time for each check. To check all three properties for an SPLDC, we need between 3 to 12 checks according to the approach outlined in Sec. IV depending on each property's level of satisfaction.

To synthesize realistic SPLDC exemplars for our study, we used existing publicly available datasets of real models. For domain models, we used the we used the class diagrams of the metamodels of the AtlanMod Metamodel Zoo [14]. For feature and choice models, we used the SPLOT feature model repository [19]. Feature models in SPLOT are expressed in a custom language for expressing logic constraints. We transformed 30 randomly selected SPLOT feature models to *Tyson* feature models. Furthermore, to get Tyson choice models, we transformed a different set of 30 randomly selected SPLOT feature models. We combined all parts with randomly generated mappings.

We then categorized the feature, choice, and domain models in three size categories: small, medium and large, as shown in Table III. Each synthetic SPLDC exemplar was then generated by combining a model from all three categories and generating random feature and decision mappings. This process resulted in 270 different synthetic exemplars, 10 in each size category combination.

For our experiments, we used 3 product-level properties, shown in Table IV. They are realistic class diagram consistency properties, inspired from published work by Van Der Straeten et al. [30].

TABLE III
NUMBER OF ELEMENTS IN EACH CATEGORY

Element Type	Small	Medium	Large
# of Features in FM	10-15	16-30	31-45
# of Choices in CM	10-15	16-30	31-45
# of Classes in DM	7-15	16-30	31-45

TABLE IV
PROPERTIES CHECKED FOR SCALABILITY ANALYSIS

#	Property
R_4	$\nexists$ Association A in DM: (A.from = none OR A.to=none)
R_5	$\nexists$ Class C in DM: $\forall$ Association A in DM (C not in A.from) AND (C not in A.to)
R_6	$\nexists$ Association A in DM: (A.from =A.to)

Following the decision procedure outlined in Sec. IV, we checked the three consistency properties shown in Table IV for each SPLDC exemplar and recorded the run times to check the scalability of the approach. To understand the effect of Alloy's *scope* in the analysis, we tried global scopes of 20, 40, and 60. The results of property analysis were same for all three scopes, however the analysis became much slower. Below, we report results for scope 40.

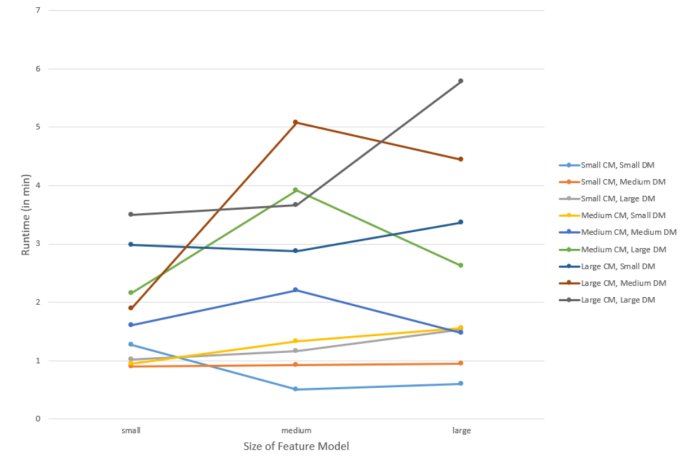
B. Results

We show the effects on the run time of the size of feature model, choice model and domain model in Fig. 6. Each of the 27 datapoints is the average of 10 runs for the respective size category combination. The maximum time for a check was approximately 6 minutes for the category with large feature models, large choice models, and large domain models.

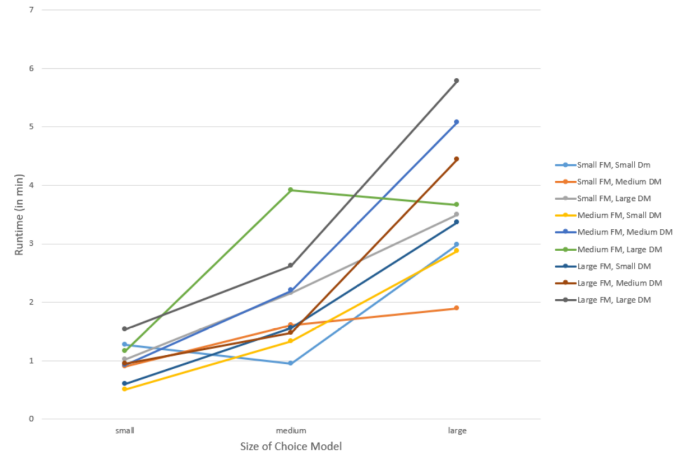
We observe in Fig. 6(b) that in most of the cases, run time increases with increase in size of choice models. Similarly, we see in Fig. 6(c) for the majority of the categories run time increases with increase in domain model size. However, there are some categories where run time decreases despite increase in domain model size. We also observe that Fig. 6(a) shows a mixed trend in run time with respect to the size of feature models: for some categories run time increases with increasing size of feature models, for others it decreases despite an increase in the size of feature model.

We note that that size of choice model has the biggest impact on run time. In other words, run time is larger for SPLDCs with more design uncertainty. The size of domain model has an anticipated impact on runtime, however, the impact of the size of feature model is not easily discernible. We attribute this to the fact that our specification first quantifies over choices and subsequently over features. This increases the chance that the space of feature combinations explored by the SAT solver has been pruned by prior design choice options.

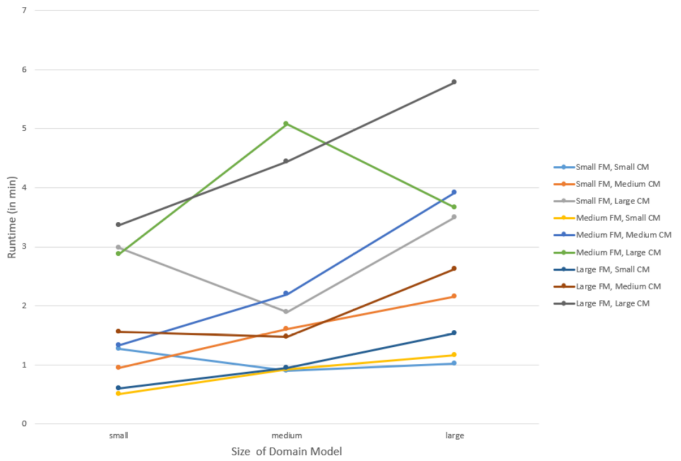
Regardless, we note that the increases in size in any of the categories do not precipitate dramatic changes in runtime. In other words, for the size of models with which we experimented, we do not see evidence that increasing the size of any of the components of an SPLDC makes the property checks



(a)



(b)



(c)

Fig. 6. Runtimes: (a) Effect of the size of feature model, for different sizes of choice model and domain model (b) Effect of the size of choice model, for different sizes of feature model and domain model (c) Effect of the size of domain model, for different sizes of choice model and feature model

infeasible. This is a preliminary result that encourages us to conduct further research on the topic.

C. Threats to Validity

One threat to validity stems from our choice of experimental subjects. To mitigate the lack of real examples of model-based SPLs, we opted to synthesize realistic, semi-random exemplars. We used real models from publicly available sources and only randomly generated the mappings. This allowed us to generate SPLDCs of various sizes, and so to explore the effect of size changes. Another threat to validity comes from our choice or properties. To mitigate this, we chose examples of properties that were inspired from published literature and represent typical structural properties of models found in MDE practice. The choice of Alloy scope may also affect the results. To mitigate this, we experimented with various scopes, validating that the choice of scope did not change the property check results. Regardless, we report observations for a high enough scope such that the slowdown effects would be observable. Finally, the sample size for each category may not have been large enough. We intend to perform a deeper evaluation with more and larger exemplars.

Regarding construct validity, we note that we experimented with choice models that modelled simple binary design choices. This is based on two assumptions that underlie this paper: (a) that modellers know what design choices they face, and (b) that for each choice they have elicited and modelled alternative solutions In other words, design uncertainty manifests itself in terms of fully enumerated "known unknowns". These assumptions do not necessarily hold in real design uncertainty scenarios. As discussed in Sec. III, in this paper, we aimed to understand whether reasoning in the presence of both variability and uncertainty about design decisions is feasible in the first place. Further research is needed to quantify the effects of partially enumerated or unknown unknowns. However, these preliminary results indicate that a formal reasoning technique could be used as part of more holistic approach to manage design uncertainty in SPLE.

VII. Related Work

The work presented here follows up on a previously published vision paper [10], that originally postulated the need for managing variability and design uncertainty at the same time. More broadly, this work deals with the capturing and management of uncertainty about design decisions among a set of possible SPLs, including at design time, a concern that has been posed by Metzger and Pohl [20] while they outline some ongoing research challenges in product line engineering based on major trends in SPLE. We build on previous work on the modelling and reasoning for design uncertainty in software models that did not consider the relationship with explicit variability management [11].

Tran and Massacci [29] studied the management of risk that is caused by uncertainty about the evolution of the feature model of an SPL. For instance, uncertainty about the features that will be implemented in the future. They proposed an approach to support decision making that takes into account the probability that individual products have of surviving over long periods of time. Three key differences between their work and ours are (a) the type of analysis, (b) the emphasis on long versus short term change, and (c) the semantics of uncertainty [7] that is being considered. The uncertainty surrounding the evolution of a feature is *aleatory* (i.e., predicated on randomness and is generally modelled using probability) because it depends on the future context in which it will take place. On the other hand, uncertainty about short term decisions is *epistemic* (i.e., predicated on lack of knowledge which is termed as uncertainty on decision maker's side) since in the short term the context is constant.

Two notable examples of research attempt to explicitly model different kinds of choices present in SPL engineering. Lytra et al. [12] proposed a framework in which decisions about variability and architectural design are conceptually differentiated and used in a synergistic way. The focus of their work is to model the two dimensions, mapping variability options to architectural design alternatives in order to capture and manage the dependencies between them, as well as for providing decision support for creating variants. Barner et al. [1] proposed a technique for design space exploration for SPLs, using evolutionary optimization with the aim of discovering alternative implementations for a given functional description of a product variant. They identify two types of configuration choices: those having to do with "business variability", and those concerning "technical variability". The former express end user functionality; the latter represent technical alternatives that can deliver the same functionality, albeit with different quality characteristics. Our work is complementary to such approaches, since our focus is on reasoning and analysis.

Finally, there exists a large body of work that concerns modelling variability and dealing with challenges of managing variability [3]. Furthermore, there exist a variety of approaches to reason about variability without generating all products of an SPL [28], e.g., by feature-oriented analysis or using variability information while analyzing. Examples of this include techniques for model checking product lines [4], for automatically lifting analyses to account for variability [21], and for formalizing and analyzing behavioural specifications of product line requirements [32]. In principle, any technique for variability analysis can be used to reason about both design and variability choices, provided that the two are modelled at the same level of abstraction (in the same feature model). However, it is useful to separate the two concerns, as design choices may affect the variability abstractions themselves. And unless we take care to model the two concerns separately, we cannot easily express properties that quantify differently the two sets of choice variables and thus generate nuanced feedback.

VIII. Conclusion

We have described a preliminary approach for modelling uncertainty about design choices in Software Product Lines

(SPLs) and for checking properties in the presence of both design uncertainty and variability.

Making two basic assumptions (that modellers know what design choices they are uncertain about and have elicited a set of alternative solutions), we have shown how to formally represent design choices in different elements of an SPL definition, resulting in "Software Product Lines with Design Choices" (SPLDCs). An SPLDC represents a two dimensional space defined by two axes: variability configurations and design decisions. In order to check the satisfaction of product-level properties of SPLDCs, we use four levels of property satisfaction, first introduced in previous work [10], which allow different quantification over design decisions and product configurations. We implemented support for SPLDC modelling in Tyson, a textual language, with semantics based on Alloy. We then showed how the satisfaction level of properties can be checked for SPLDCs expressed in Tyson. We studied the efficiency of reasoning, finding that property checking for SPLDCs comes at a computational cost, with the size of choice models being the most important determining factor for the runtime of verification.

The feedback provided by our approach includes either a feature configuration or a design decision. However, it does not indicate which of the features or choices in the counterexample are specifically responsible for the violation of the property. As evident from the scalability study, there is a large computational complexity for property checking. This can hinder developers from reasoning about extra large SPLDCs. Lastly, our approach is focused on structural properties of models; behavioural properties of SPLDCS remains future work.

In the future, we aim to further evaluate our approach with a large case study from the domain of real-time reactive embedded systems. We also intend to further investigate the use of other reasoning engines such as Alloy* [22] and alternative automated reasoning formalisms, such as QBF solving [17] with the aim to improve the efficiency of our implementation. We want to expand Tyson to support various other representations of domain models, in order make it applicable to a wider range of problems. We also want to expand the expressiveness of the decision model, to allow representing more complex design decisions, such as those involving numeric ranges. Finally, we want to expand our approach to the analysis of requirements expressed as behavioral properties.

References

[1] Barner, S., Diewald, A., Eizaguirre, F., Vasilevskiy, A., Chauvel, F.: Building product-lines of mixed-criticality systems. In: Proc. of FDL'16, pp. 1–8 (2016)

[2] Björner, A., Edelman, P.H., Ziegler, G.M.: Hyperplane arrangements with a lattice of regions (1990)

[3] Chen, L., Babar, M.A., Ali, N.: Variability management in software product lines: a systematic review. In: Proc. of SPLC'09 (2009)

[4] Classen, A., Heymans, P., Schobbens, P.Y., Legay, A., Raskin, J.F.: Model checking lots of systems: Efficient verification of temporal properties in software product lines. In: Proc. of ICSE'10, pp. 335–344 (2010)

[5] Czarnecki, K., Antkiewicz, M., Kim, C.H.P., Lau, S., Pietroszek, K.: Model-driven software product lines. In: Companion to Proc. of OOPSLA '05, pp. 126–127 (2005)

[6] Czarnecki, K., Grünbacher, P., Rabiser, R., Schmid, K., Wasowski, A.: Cool features and tough decisions: A comparison of variability modeling approaches. In: Proc. of VaMoS '12, pp. 173–182 (2012)

[7] Esfahani, N., Malek, S.: Uncertainty in Self-Adaptive Software Systems. In: R. de Lemos, H. Giese, H. Müller, M. Shaw (eds.) Software Engineering for Self-Adaptive Systems 2, Lecture Notes in Computer Science Hot Topics (2012)

[8] Eysholdt, M., Behrens, H.: Xtext:implement your language faster than the quick and dirty way. In: Companion to Proc. of OOPSLA'10 (2010)

[9] Famelis, M., Chechik, M.: Managing design-time uncertainty. Software & Systems Modeling (2017)

[10] Famelis, M., Rubin, J., Krzysztof, Salay, R., Chechik, M.: Software Product Lines with Design Choices: Reasoning about Variability and Design Uncertainty. In: Proc. of MODELS'17, pp. 93–100 (2017)

[11] Famelis, M., Salay, R., Chechik, M.: Partial Models: Towards Modeling and Reasoning with Uncertainty. In: Proc. of ICSE'12, pp. 573–583 (2012)

[12] I. Lytra and H. Eichelberger and H. Tran and G. Leyh and K. Schmid and U. Zdun: On the interdependence and integration of variability and architectural decisions. In: Proc. of VaMOS'14 (2014)

[13] Jackson, D.: Software Abstractions: logic, language, and analysis. MIT press (2012)

[14] Jouault, F., Bézivin, J.: Km3: A dsl for metamodel specification (2006)

[15] Kästner, C., Apel, S.: Integrating Compositional and Annotative Approaches for Product Line Engineering. In: Proc. of GPCE'08, pp. 35–40 (2008)

[16] Kaur, N.: Modelling and reasoning with software product lines with design choices. Master's thesis, Université de Montréal (2019)

[17] Klieber, W.: Formal Verification Using Quantified Boolean Formulas (QBF). Doctor of philosophy (2014)

[18] Le Berre, D., Parrain, A.: The SAT4J library, Release 2.2, System Description. Journal on Satisfiability, Boolean Modeling and Computation **7**, 59–64 (2010)

[19] Mendonca, M., Branco, M., Cowan, D.: SPLOT: software product lines online tools. In: Companion to Proc. of OOPSLA'09, pp. 761–762. ACM (2009)

[20] Metzger, A., Pohl, K.: Software Product Line Engineering and Variability Management: Achievements and Challenges. In: Proc. of FOSE'14, pp. 70–84 (2014)

[21] Midtgaard, J., Dimovski, A.S., Brabrand, C., Wasowski, A.: Systematic derivation of correct variability-aware program analyses. Science of Computer Programming **105**, 145 – 170 (2015)

[22] Milicevic, A., Near, J.P., Kang, E., Jackson, D.: Alloy*: a general-purpose higher-order relational constraint solver. Formal Methods in System Design (2017)

[23] Obeo: Acceleo (2017). URL http://www.eclipse.org/acceleo/

[24] Pohl, K., Böckle, G., Linden, F.J.v.d.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag, Berlin, Heidelberg (2005)

[25] Ramirez, J., Jensen, C., Cheng, H.: A taxonomy of uncertainty for dynamically adaptive systems. In: Proc. of SEAMS'12 (2012)

[26] Schobbens, P.Y., Heymans, P., Trigaux, J.C.: Feature diagrams: A survey and a formal semantics. In: Proc. of RE'06, pp. 139–148 (2006)

[27] Seidl, C., Schaefer, I., Assmann, U.: Capturing variability in space and time with hyper feature models. In: Proc. of VaMoS '14, pp. 6:1–6:8 (2013)

[28] Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G.: A Classification and Survey of Analysis Strategies for Software Product Lines. ACM Comput. Surv. **47**(1), 6:1–6:45 (2014). DOI 10.1145/2580950

[29] Tran, L.M.S., Massacci, F.: An approach for decision support on the uncertainty in feature model evolution. In: Proc. of RE'14 (2014)

[30] Van Der Straeten, R., Mens, T., Simmonds, J., Jonckers, V.: Using description logic to maintain consistency between uml models. In: Proc. of UML'03, pp. 326–340 (2003)

[31] Weaver, S., Franco, J., Schlipf, J.: Extending Existential Quantification in Conjunctions of BDDs. J. on Satisfiability, Boolean Modeling and Computation **1**, 89–110 (2006)

[32] Zhou, J., Lu, Y., Lundqvist, K., Lönn, H., Karlsson, D., Liwang, B.: Towards feature-oriented requirements validation for automotive systems. In: Proc. of RE'14 (2014)