Software Product Lines with Design Choices: Reasoning about Variability and Design Uncertainty

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

Julia Rubin
University of British Columbia
julia.rubin@ubc.ca

Krzysztof Czarnecki
University of Waterloo
kczarnec@gsd.uwaterloo.ca

Rick Salay, Marsha Chechik
University of Toronto
{salay,chechik}@cs.toronto.edu

***Abstract*—When designing changes to a software product line (SPL), developers are faced with uncertainty about deciding among multiple possible SPL designs. Since each SPL design encodes a set of related products, dealing with multiple designs means that developers must reason about sets of sets of products. The additional degree of multiplicity is not well described by existing product line abstractions. In this paper, we propose an approach for dealing with design uncertainty within SPLs using a novel composition of variability modelling with an abstraction for capturing and managing design uncertainty. This allows developers to accurately describe the decisions involved in making changes to an SPL during the design stage and provides them with a framework for SPL design space exploration by analyzing and enforcing SPL properties.**

I. Introduction

The increasing complexity of large software-intensive system development has lead to the adoption of Software Product Line Engineering (SPLE). SPLE aims to support developers in managing the variability within sets of software product variants that are similar but different [1], [2], [3]. In SPLE, developers typically explicate *variation points* (a.k.a. *features*) and relationships between them in a feature model. A selection of features from this model guides the derivation of a specific product of a Software Product Line (SPL). Within the SPLE framework, the SPL representation is the primary development artifact used for tasks such as automated analysis [4], configuration [5], transformation [6], and others [7].

SPLE allows organizations to make long-term commitments to the maintenance of families of related products. However, when designing changes to an SPL, engineers need to express and reason about short-term design choices, which form a *design space*. Such choices are a source of *design uncertainty* [8] and can result from dealing with different design alternatives [9], making decisions about architecture [10], resolving inconsistencies [11], or resolving conflicting stakeholder requirements [12]. The resulting combination of variability and design uncertainty involves reasoning about a set of possible SPLs, each of which is, by itself, a set of software products, as shown in Figure 1.

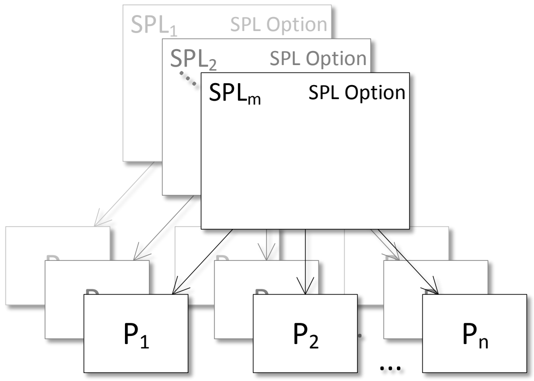


Fig. 1: Sets of sets of products.

The space of possibilities induced by design uncertainty is driven by goals different from those of variability management. Variability management is concerned with supporting different variants of software that serve multiple customers or market segments [1]; uncertainty in design space management is concerned with exploring and assessing alternatives to ultimately make informed design decisions [13]. Design uncertainty is an aspect of the development process itself; it is transient and must be reduced and eventually eliminated as knowledge is gathered and decisions are made. Thus, the ultimate goal of resolving design uncertainty is to produce *one desired artifact*. On the other hand, variability is an aspect of the artifacts simultaneously managed through the entire development process; it is to be preserved and carefully engineered to represent the desired range of product variants. Thus, product lines aim to produce and simultaneously manage *multiple desired artifacts*.

Any technique for managing either variability or design uncertainty in isolation should provide ways of encoding and managing *sets* of artifacts. Therefore, techniques for representing, reasoning with and manipulating these sets naturally have a lot in common. Yet, simply reusing variability abstractions is not enough since the goals for creating each set are quite different and thus lead to distinct methodological considerations. We focus on *property-guided SPL design space exploration*. In this context, for example, a developer might want to ensure that some property can be guaranteed to be satisfied by at least one product of every possible product line design, or study the effect of a design decision across all possible products in the SPL design space. If a property is found to be violated due to a design decision, the property can be enforced by simply pruning the design space. If however the violation is caused by a variation point, a more in depth investigation might be needed to salvage a faulty (but desirable) variant.

In this paper, we look at the problem of expressing and reasoning about *design uncertainty within SPLs*, and supporting the SPL design space exploration process. Our goal is to understand the synergy and complementary usage of variability and design uncertainty and to demonstrate how they can be meaningfully leveraged together. To do this, we combine variability abstractions with *partial modelling*, a technique for managing design uncertainty within a software model [14], [15]. A partial model explicates *decision points* and represents the set of possible models that could be obtained by making decisions and resolving design uncertainty. In turn, partial models can be used as a primary development artifact for tasks such as automated reasoning [15], transformation [16] and refinement [17].

We identify the categories of properties that can be specified and checked in SPLs with design uncertainty, as well as strategies for resolving possible violations of these properties. We show that property violations are better understood when variability and design uncertainty are viewed separately. Additionally, we demonstrate that the conceptual separation allows us to guide developers to explore the space of SPL designs by planning appropriate responses to property violations. We thereby highlight the difference and the synergy between SPLE and partial modelling, and allow system development that leverages their combination.

We make the following contributions: 1) We define *Software Product Lines with Design Choices* (SPLDCs), which combine SPLs with partial models, thus showing how variability and design uncertainty can be conceptually distinguished and modelled. 2) We introduce four categories of SPLDC properties that allow quantification over the space of designs and variants. 3) We describe how to use SPLDCs to explore the SPL design space using strategies for responding to SPLDC property violations. These contributions allow us to outline open research questions and directions for future work.

The remainder of the paper is organized as follows. We discuss related work in Section II. In Section III, we define notations for expressing SPLs and models with design uncertainly, and demonstrate the combination of the two by presenting a concrete example of a set of SPL alternatives. In Section IV, we identify four categories of properties that can be defined about SPLs with design uncertainty, and, in Section V, show how responding to violations of these properties can be used to guide the SPL design exploration process. We discuss the next steps towards effectively supporting SPLDCs and outline the main challenges in Section VI, and conclude the paper in Section VII.

II. Related Work

The research question about how to reason in the presence of both variability and uncertainty has recently been posed by Metzger and Pohl [18]. Our approach is a step in that direction, focusing specifically on *design-time* uncertainty, i.e., the uncertainty encountered in the case where a designer does not have enough information to choose from a set of design alternatives [15]. We additionally assume (a) that the SPL designers are aware of the relevant unresolved design decisions, and (b) that they have elicited a set of possible SPL designs for each decision.

The similarity between automated techniques for handling variability and design spaces has been pointed out before. For example, Neema et al. [19] noted that "design space modeling is essentially creating product line model architectures". Mjeda et al. [20] compared SPLE with domains that deal with the management of sets of artifacts: Multiple Criteria Decision Analysis (MCDA), Multi-Objective Optimisation (MOO), and Design Space Exploration (DSE). Despite the fact that they defined DSE narrowly as a problem of identifying how to deploy embedded software on hardware controllers, they were able to identify correspondences in artifacts and tasks between the different fields. This confirms our claim that design space management and variability management share many similarities; we propose to further take advantage of these similarities to provide methodological support for design in SPLE.

Related is the problem of deciding the right binding time for a particular variability point [21]. Bosch et al. [22] pointed out that variability points may appear at various times during the development of an SPL. In one extreme, binding can take place at runtime, e.g., in the field of dynamically adaptive systems (DASs) [23], which are often understood as dynamic product lines [24]. In DASs, run-time decisions help the system adapt to changes in the environment in which it operates. In the other extreme, binding can take place at design time. The need to model the difference between the two extremes has also been acknowledged in the DAS community, where Ramirez et al. [8] have identified several sources of design-time uncertainty for DASs for which no mitigation strategy currently exists. A conceptual distinction between product variants and design alternatives supports more precise exploration of the space of SPL designs (see Section V).

In published literature, the distinction between variability and design uncertainty is rarely made, with a notable exception of the work by Lytra et al. [25], where decisions about variability and architectural design are conceptually differentiated and used synergistically. Lytra et al. further discuss related work in SPL engineering to demonstrate that the two concerns are not treated separately. The main focus of their work is on modelling the two spaces and making explicit their interdependencies by mapping variability choices to architectural alternatives. Our approach is complementary, as we aim to guide the SPL design process by analyzing the properties of the design space and using the analysis results to develop strategies for exploring it.

III. Modeling SPL Alternatives

In this section we review existing approaches for representing SPLs (Section III-A) and then introduce SPLDCs (Section III-B) and their logical representation (Section III-C).

A. Modelling SPLs

Figure 2 shows an simple example product line of washing machine controllers [26] expressed as a UML state machine. Ignore the dashed lines for the time being. The *feature model*

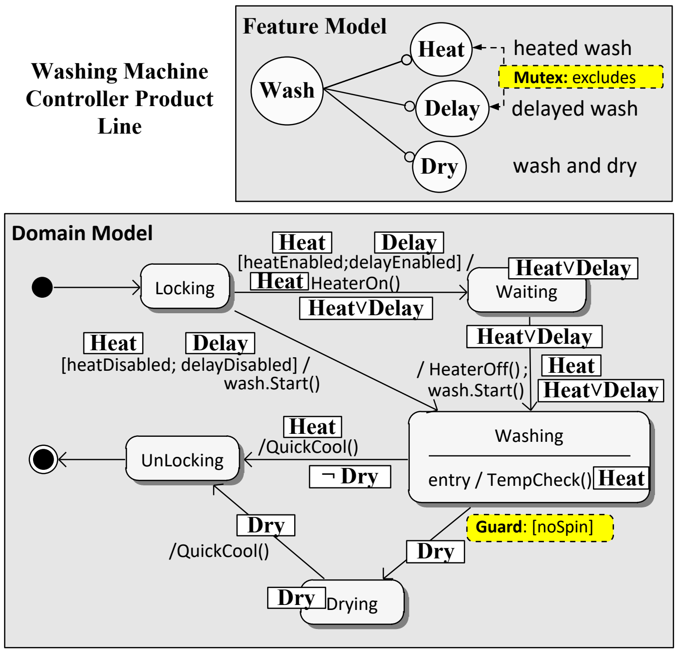


Fig. 2: Example washing machine controller product line with uncertainty $\mathcal{W}$. Decision points are shown using dashed lines.

shown at the top specifies that the basic **Wash** feature can be optionally augmented with three additional features: **Heat** supports heating water in case there access only to the cold water supply, **Delay** allows setting a future time to start the wash, and **Dry** adds the capability to also function as a dryer. The *domain model* shown at the bottom is a state machine that describes the progression of a washing cycle: after locking and initialization, the machine waits until the water is heated or until the specified delay time has elapsed, or skips directly to the main washing phase; this is followed by an optional drying phase or skips directly to unlocking and finalization. Various elements of the domain model are annotated with *presence conditions* [27] indicated in bold. These are formulas that specify what configurations of features enable a particular element. For example, the guard `heatEnabled` on the transition between states `Locking` and `Waiting` is only present in configurations that have the **Heat** feature.

B. Modelling SPLs with design uncertainty

Partial models [15] allow developers to express their design uncertainty about any element of the model, supplementing it with a "May Formula" that captures dependencies between the individual choices. We call the combination of SPLs and partial models "Software Product Lines with Design Choices" (SPLDCs). For our scenario, design uncertainty can apply to any part of an SPL definition – a domain model, a feature model, or presence conditions. In the example that follows, we illustrate design uncertainty in the first two.

The washing machine SPLDC has two decision points capturing designer uncertainty, indicated in Figure 2 using dashed lines. The decision **Mutex**: *Should the features **Heat** and **Delay** be mutually exclusive?* is modeled by annotating the `Excludes` constraint arrow in the feature model with a propositional variable encoding whether the arrow should be included in the model. Answering "no" to the decision **Mutex** thus corresponds to setting the propositional annotation of `Excludes` to False, and thus removing it from the model. Similarly, the decision point **Guard**: *Should the single incoming transition to the state* `Drying` *be guarded by a check that the drum is not spinning?* is modeled by annotating the guard `noSpin` in the domain model. In our example, the two decision points are independent of each other and, therefore, the May Formula is trivially true.

Since both **Mutex** and **Guard** are binary decisions that are mutually independent, the resulting design space $\mathcal{W}$ consists of four possible product lines, each of which in turn represents a set of possible products. Each product line in $\mathcal{W}$ can be *derived* by making a set of concrete decisions about **Mutex** and **Guard**. For example, a product line W_{d1} can be derived using the decision set $d_1 = \{Y, N\}$, where **Mutex** is answered Yes (i.e., the `Excludes` arrow is included in the feature model) and **Guard** is answered No (i.e., the guard `noSpin` is absent from the incoming transition of the state `Dry`). The product line W_{d1} can be configured to produce six individual washing machines; moreover, neither of the three products that have **Dry** have the `noSpin` guard.

C. Semantics of SPLDCs

Czarnecki and Pietroszek [28] showed how to encode an SPL representation with propositional presence conditions and the feature diagram into a propositional formula so that its valuations correspond to the valid configurations of the product line. Famelis et. al. [15] showed how to encode a model with design uncertainty (i.e., a partial model) into a propositional formula whose valuations correspond to the possible models produced by different combinations of design decisions. Putting the two encodings together allows us to encode the entire SPLDC into a single propositional formula Φ that contains decision variables, feature variables and additional variables to encode the domain model. A particular value assignment to the decision variables identifies a product line in the design space, and a further assignment to the feature variables identifies a product within the product line. Formula Φ evaluates to true only when a valid assignment to the decision variables and feature variables is given.

For example, the following fragment of $\Phi^{\mathtt{WM}}$ for the washing machine example encodes the feature model along with decision point **Mutex**:

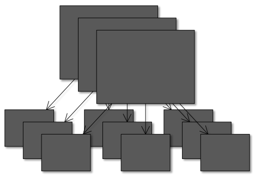
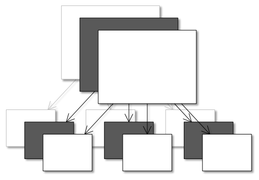
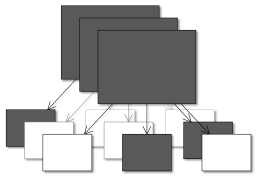
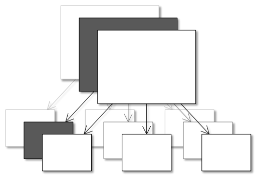
$$\ldots f_{\mathtt{Wash}} \wedge (d_{\mathbf{Mutex}} \Rightarrow (f_{\mathtt{Heat}} \veebar f_{\mathtt{Delay}})) \ldots$$

Here, f_x and d_y are the propositional variables corresponding to feature variables and decision points, respectively. The fragment encodes that **Heat** and **Delay** are mutually exclusive when **Mutex** is true.

IV. Properties of SPLs with Design Uncertainty

In this section, we aim to describe the properties of SPLDCs. Such properties may represent product line requirements, engineering constraints, "what-if" scenarios, etc.

TABLE I: Categories of SPLDC-level properties and possible responses to their violations.

	All Products		**Some Products**	
	Necessary for product line	**Possible for product line**	**Necessary for product line**	**Possible for product line**
				
Name	$C_{NA}(R)$	$C_{PA}(R)$	$C_{NS}(R)$	$C_{PS}(R)$
Description	Property R must hold in every product of every product line	Property R must hold in every product of at least one product line	Property R must hold in at least one product of every product line	Property R must hold in at least one product of at least one product line
Formalization	$\forall \bar{d} \cdot \Phi_D \Rightarrow \left(\forall \bar{f} \cdot \Phi \Rightarrow R\right)$	$\exists \bar{d} \cdot \Phi_D \wedge \left(\forall \bar{f} \cdot \Phi \Rightarrow R\right)$	$\forall \bar{d} \cdot \Phi_D \Rightarrow \left(\exists \bar{f} \cdot \Phi \wedge R\right)$	$\exists \bar{d} \cdot \Phi_D \wedge \left(\exists \bar{f} \cdot \Phi \wedge R\right)$
Response strategies	Reduce design uncertainty, reduce variability, relax constraint, or change constraint to any other type	Expand design uncertainty, reduce variability, relax constraint, or change constraint type to $C_{PS}(R)$	Reduce design uncertainty, expand variability, relax constraint, or change constraint type to $C_{PS}(R)$	Expand design uncertainty, expand variability, or relax constraint

A. *Product line properties*

Given an SPL, we can define *product-level* and *feature-level* properties. Feature-level properties involve reasoning about the feature model itself, e.g., "every mandatory feature is decomposed to optional features". We do not consider such properties in this paper. Product-level properties are properties which can hold or fail in an individual product. In the washing machine example, consider:

R1: State Unlocking must always be reachable.
R2: Action TempCheck() is called before entering Drying.

Clearly, property R1 must hold in *every* washing machine variant because a washing machine that may never unlock is unacceptable. On the other hand, assume that the manufacturer wants to be able to market an "eco" washing machine that smartly adjusts the drying temperature, and property R2 is required for such a variant. Thus, our product line should contain *only some* product satisfying R2.

B. *SPLDC-level properties*

When reasoning about SPLDCs, we would like to either consider whether the property holds for the entire design space, i.e., such a property is "required", or we would like to ensure that it is "possible" to satisfy the property in a portion of the space because we are not yet certain whether we need it and would like to keep our options open. For example, the property R1 is required for all product lines, regardless of how decision **Guard** is resolved. On the other hand, if the marketing department is still undecided whether the eco washing machine is required, the developers want to ensure that they do not commit to a design decision that would make property R2 impossible to satisfy.

Given a product-level property R, i.e., a property defined on a single product, we define four categories of *SPLDC-level properties*: $C_{NA}(R)$ ("necessary all"), $C_{PA}(R)$ ("possible all"), $C_{NS}(R)$ ("necessary some"), and $C_{PS}(R)$ ("possible some"). We summarize them in Table I. For example, since we are certain that property R1 should hold in each product of each product line, it is a *necessary all products* property, C_{NA}(R1), i.e., all products of all product lines must satisfy R1. On the other hand, if marketing is still undecided if an eco washing machine is required, we express it with a *possible some products* property, C_{PS}(R2), requiring that there be some product lines containing an R2 product. We illustrate the meaning of each SPLDC-level property category in the first row of Table I, using black boxes to indicate product line designs and individual products that satisfy the property.

C. *Expressing SPLDC-level properties*

Given an SPLDC whose design space is encoded as a propositional formula Φ and a product-level property expressed as a propositional formula R, we can capture the four categories of SPLDC-level properties in first-order logic, quantified over decision points and features. Let $\bar{\mathbf{d}}$ be the set of propositional *decision variables*, each corresponding to a decision point in the product line with uncertainty. For the washing machine example, we have $\bar{\mathbf{d}}^{\texttt{WM}} = \{d_{\textbf{Mutex}}, d_{\textbf{Guard}}\}$. Let $\bar{\mathbf{f}}$ be the set of propositional *feature variables*, each corresponding to a feature in the product line with design uncertainty. For the washing machine example, we have $\bar{\mathbf{f}}^{\texttt{WM}} = \{f_{\textbf{Wash}}, f_{\textbf{Heat}}, f_{\textbf{Delay}}, f_{\textbf{Dry}}\}$.

The formula Φ only evaluates to true for certain combinations of decision and feature variables. Let Φ_D be the dependency constraint on just the decision variables that is derived from Φ by *quantifying out* [29] all variables except for decision variables. This allows us to separate quantification over decision and feature variables. Each combination of values assigned to the decision variables is a set of decisions in the design space that define a single SPL design. Each subsequent combination of feature variables defines a product of that SPL design. In the washing machine example, we get $\Phi_D^{\texttt{WM}}$ by quantifying out the variables $\bar{\mathbf{d}}^{\texttt{WM}}$ from $\Phi^{\texttt{WM}}$, defined in Section III-C. The decision points **Mutex** and **Guard** are independent to each other, so $\Phi_D^{\texttt{WM}}$ reduces to the formula true.

Each of the four categories is encoded as a quantification first over the decision variables (i.e., individual product lines) in the design space and then over individual products (quantified over feature combinations). This is shown in the Formalization row of Table I. For example, the property R2 can be expressed by saying that a state on every path to state

Drying must have an entry action TempCheck. We define the helper propositional variable *entry*(x,y) that is true when y is an entry action of a state x. Thus, we can encode the property R2 as: $\texttt{R2} := \textit{entry}(\texttt{Locking}, \texttt{TempCheck}) \vee \textit{entry}(\texttt{Washing}, \texttt{TempCheck})$. Then, using the template for C_{PS} from the fourth column of Table I with $\bar{\mathbf{d}}^{\texttt{WM}}$ and $\bar{\mathbf{f}}^{\texttt{WM}}$ as defined above, we write: $C_{PS}(\texttt{R2}) = \exists d_{\textbf{Mutex}}, d_{\textbf{Guard}} \cdot \Phi_D^{\texttt{WM}} \wedge (\forall f_{\textbf{Wash}}, f_{\textbf{Heat}}, f_{\textbf{Delay}}, f_{\textbf{Dry}} \cdot \Phi^{\texttt{WM}} \wedge \texttt{R2})$.

V. Responding to Property Violations

We can use the encoding of an SPLDC to automatically check its properties. If this check fails, we can plan appropriate responses. We consider the following response strategies: (a) relaxing the product-level property, (b) changing the category of the SPLDC-level property, (c) decreasing the degree of design uncertainty (i.e., pruning the SPL design space), (d) increasing the degree of design uncertainty (i.e., expanding the SPL design space), (e) decreasing the degree of variability (i.e., removing configuration options), and (f) increasing the degree of variability (i.e., adding configuration options). Not all response strategies are applicable in every scenario. We summarize the possibilities for each SPLDC-level property in the bottom row of Table I. We discuss and illustrate property-related strategies in Section V-A and strategies that adjust the degree of variability or design uncertainty in Section V-B. In Section V-C we describe what feedback should be generated from the verification of SPLDC properties to help users select the appropriate strategy.

A. Adjusting the property

Relaxation of the product-level property is appropriate if we can determine that we have expressed too strong of a constraint. For example, we may decide that R2 should instead be just that TempCheck() is invoked, regardless of whether this happens prior to entering Drying. Since this is a trivial solution, we do not discuss it further.

Changing the category of the SPLDC-level property is reasonable if we can determine that it over-constrains the design space. Thus, we can reconsider the purpose of the constraint and change its category to a less restrictive one. For example, assume we are checking $C_{NA}(\texttt{R2})$. This constraint is violated since when **Heat** is not selected, no state has entry action TempCheck. We could relax it to $C_{NS}(\texttt{R2})$ – still require that all product lines have an "eco" product but not require that all products in a product line be "eco". This constraint is given by:

$$\forall d_{\textbf{Mutex}}, d_{\textbf{Guard}} \exists f_{\textbf{Wash}}, f_{\textbf{Heat}}, f_{\textbf{Delay}}, f_{\texttt{Dry}} \cdot \Phi^{\texttt{WM}} \wedge \texttt{R2}$$

Recall that $\Phi_D^{\texttt{WM}}$ reduces to true. The property $C_{NS}(\texttt{R2})$ is satisfied since the state **Washing** contains the entry action TempCheck when feature **Heat** is selected, and this feature can always be selected in a product line regardless of what decisions are made for **Mutex** and **Guard**.

B. Adjusting variability or design uncertainty

If we are confident about the product-level property and its SPLDC-level category, then it is appropriate to respond to a violation by adjusting the degree of design uncertainty and/or variability. In the case of $C_{NA}(\texttt{R2})$, we can respond by *reducing variability*, such as by changing the feature **Heat** from optional to mandatory. Assume we chose this intervention but also added a new constraint R3 requiring that a product with the **Delay** feature should exist in every product line (i.e., that **Delay** is not a dead feature). Now $C_{NA}(\texttt{R3})$ is violated because if **Mutex** $=$ true and **Heat** is always selected (due to being mandatory), the feature **Delay** is always excluded. In this case, it would be appropriate to respond by *reducing design uncertainty* by making a decision about **Mutex**, that the two features should not be mutually exclusive. In other cases, instead of reducing design uncertainty or variability, the appropriate response is to expand them. For example, consider a constraint $C_{PA}(\texttt{R4})$ requiring that the next state of Washing is always Unlocking. Such a constraint inadvertently poses a question whether **Dry** should exist or not (regardless of its optionality). Thus an appropriate response is to expand design uncertainty by annotating **Dry** in the feature model with a new decision point **DryExists**.

C. Generating feedback

In the cases where selecting among the available response options is difficult, it is necessary to provide developers with more nuanced feedback. For this, we leverage the clear separation between design uncertainty and variability and the fact that existing reasoning techniques [15], [7] rely on the generation of witnesses to prove or disprove properties. Thus, when a violation is discovered, instead of a Boolean answer, we produce a tuple $\langle u, v\rangle$, where $u, v \in \{T$: *"true for all"*, F: *"true for none"*, S: *"true for some"*$\}$ represent the separate answer for design uncertainty and variability, respectively. This allows us to focus our response. For example, the violation of $C_{NA}(\texttt{R2})$ produces the answer $\langle F, S\rangle$, indicating that all product lines in the design space have some product that violates the property. Thus, we can determine that from the many response strategies for the SPLDC-level property category C_{NA} in Table I, in this case we can either reduce variability or change the constraint type to C_{NS}. Such a precise feedback is made possible by the separate quantification over design uncertainty and variability.

There is a tradeoff between such nuanced feedback and cost. Generating counterexamples for each dimension is equivalent to existentially quantifying Φ over $\overline{d}$ and $\overline{f}$ and then checking for satisfiability, and so is computationally expensive.

VI. Towards SPLDC Support

In this section, we describe steps towards creating effective support for working with SPLDCs, and identify the most important challenges that are to be addressed by future work.

A. Modelling SPLDCs

The creation of SPLDCs requires tooling capabilities for expressing both variability and design uncertainty. Users should be able to model variability using standard feature-based SPL abstractions and methodologies as described in Section III-A, as well as be able to model design uncertainty by identifying decision points and expressing their dependencies, as described in Section III-B.

Variability modelling can be done using any of the several tools for modelling and managing feature-based SPLs [30]. However, to also support modelling design uncertainty, the tools should support plugging in the necessary abstractions and notations for design uncertainty. Two tools in particular offer the most promise: Clafer [31] and PEoPL [32]. Clafer is a textual lightweight structural modeling language that natively supports variability abstractions, while also providing general-purpose specification and reasoning capabilities. These can be leveraged to define the infrastructure for modelling and reasoning about SPLDCs using the formal representations of SPLDCs and SPLDC properties in Section III and Section IV. PEoPL is a projectional [33] integrated development environment (IDE) for SPLE, based on the Jetbrains MPS platform [34]. MPS is a language workbench that allows plugging-in modules that define purpose specific modelling languages and their associated analysis techniques [35]. Enabling SPLDC support for PEoPL would therefore require the definition of an MPS design uncertainty language module, along with the relevant SPLDC reasoning techniques.

An alternative approach is to enhance MU-MMINT, an Eclipse-based IDE for managing design uncertainty in software models [36], with SPL capabilities. MU-MMINT supports the creation, analysis, refinement, and transformation of partial models. It is based on the MMINT interactive model management workbench [37], and thus can be extended with any modelling language defined using the Eclipse Modelling Framework (EMF) [38]. Therefore, adding SPLDC capabilities to MU-MMINT would entail integrating MMINT with Eclipse-based SPLE toolkits, such as FeaturePlugin [39] or FeatureIDE [40].

In all these cases, the main challenge is the technological integration of the different tools and the creation of intuitive work environments that allow users to seamlessly express design uncertainty during the course of their work.

B. Reasoning about SPLDC-level properties

In order to analyze the properties of an SPLDC, users should be able to (a) define product-level properties, (b) use them to create SPLDC-level properties by using the categories outlined in Table I, (c) verify the properties of the SPLDC, and (d) get feedback from the verification.

In this paper, we do not prescribe any single analysis technique. Users should be able to reason about any product-level properties for which there exists an SPL analysis technique [7], e.g., well-formedness checking [41], model checking [42], non-functional analysis [43], etc. Therefore, we do not make any assumptions about the language in which product-level

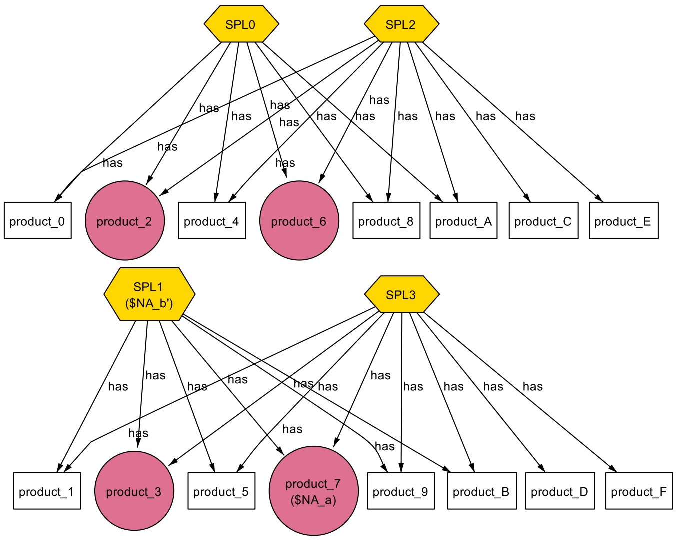


Fig. 3: Feedback for the check of C_{NA}(R2) generated by Alloy. Products violating R2 are indicated as red circles.

properties should be written since the choice of language depends on the desired analysis. In the washing machine example, R1 is a reachability property and thus might be expressed in a language such as PROMELA [44]. In Section IV-C, we argued that R2 can also be encoded structurally, and therefore might be expressed in a language such as OCL [45]. Properties can also be expressed and checked in more general-purpose languages such as Alloy [46] or Clafer [31]. Given a desired analysis technique and language, SPLDC-level properties can be encoded using the templates in Table I.

The main challenge is in adapting the various analysis techniques so that they provide separate verification feedback for variability and design uncertainty, as described in Section V-C. To better illustrate the required tool support for such feedback, we created a prototype proof-of-concept implementation using Alloy [46]. Our prototype allows encoding the SPL design space, checking SPLDC-level properties and generating feedback. We used it to represent the design space of the washing machine example. We hard-coded the evaluation of the product-level properties R1-R4 for each product in the space. Finally, we used the SPLDC-level property categories in Table I to explore the washing machine SPLDC design space.

For example, Figure 3 shows the feedback generated for the check of SPLDC-level property C_{NA}(R2). The design space consists of four SPL designs, each represented by a yellow hexagon. Each SPL design is linked to its products via the `has` arrows. White rectangles represent products for which the product-level property R2 holds and red circles those for which it does not.

Our implementation provides feedback about why the check C_{NA}(R2) fails by indicating the labels `$NA_b`' and `$NA_a` on the elements `SPL1` and `product_7`, respectively. These are automatically generated by Alloy to indicate that `SPL1` and `product_7` are witnesses for the violation of C_{NA}(R2) for variability and design uncertainty, respectively. This is possible because Alloy allows us to express and analyze any first-order logical theory in a bounded scope. However, this cannot be generalized to real SPLs where the number of features and the number of possible variants can be very large. The challenge is therefore to adapt SPL analysis techniques to be able to treat design uncertainty separately. This includes finding the right balance between expressiveness and analytical power.

C. Planning responses

Assuming that appropriate feedback can be generated for property violations, users should be supported with guidance for selecting among the response strategies listed in Table I. The main challenge is in developing comprehensive methodological support for selecting responses to property violations. This includes: assessing whether the category of a SPLDC-level property is too restrictive, identifying appropriate relaxations for product-level properties, and providing guidance for reasonable edits to an SPLDC to adjust the degree of variability or design uncertainty.

For example, using the feedback generated from our prototype Alloy implementation for the violation of the property $C_{NA}(\texttt{R2})$ that is shown in Figure 3, the user can identify that pruning the design space by removing the design `SPL1` is not sufficient since the offending product (`product_7`) can also be derived from the design `SPL3`. In fact, each product line design contains a product that violates `R2`. Thus, reducing variability (i.e., removing configuration options) is a better way to respond to the violation. In this case, an important challenge is to support the evolution of SPLDCs by recommending specific edits to the feature model that ensure that unwanted configurations are excluded. However, simply deleting products from the SPLDC may not be acceptable, and developers may choose to fix them instead. One challenge is therefore to help users to assess the configurations resulting in products that violate a property of interest. Another challenge is to support users with recommendations about how to repair the problematic products of the SPLDC.

Another important challenge is to provide guidance to users in the case where the appropriate response to a property violation is to either add configuration options (thus increasing the degree of variability), or to expand the design space by increasing design uncertainty. This could be accomplished by helping users identify candidate SPL designs or products for which small changes are required to satisfy a SPLDC property.

VII. Summary

Software Product Line Engineering allows the long-term maintenance of set of related software systems by modelling their commonalities and variabilities. However, during the design of changes to an SPL, developers are faced with short-term design uncertainty about managing a design space of possible SPL designs. Since each SPL design encodes a set of related products, dealing with multiple designs means that developers must reason about sets of sets of products. This combination of variability and design uncertainty is not well supported by existing SPL engineering techniques.

We have proposed an approach for combining SPLs with partial models, a formalism for representing design uncertainty. The resulting artifacts are called Software Product Lines with Design Choices (SPLDCs). The integration of variability and design uncertainty in a common formalism helps developers to conceptually differentiate between the kinds of decisions that are relevant during the design and configuration stages of the SPL lifecycle. This enables developers to create better descriptions of the design space of SPLs, to reason about its properties and to get feedback that is useful for exploring it. We have described the various properties that can be expressed about SPLDCs by quantifying separately over features and decision points, resulting in four SPLDC property categories. We have shown how to logically encode such properties and we described what feedback should be produced by tools supporting SPLDC analysis. Based on this feedback, we outlined possible response strategies in the case of property violation. These strategies provide a framework within which to explore the design space of SPLs that is encoded by SPLDCs.

We have outlined the next steps in developing support for working with SPLDCs and identified the major challenges. To effectively support the creation and management of SPLDCs, we must augment existing SPLE tools with techniques for design uncertainty. To realize SPLDC analysis techniques, we must adapt SPL reasoning techniques so that they can handle design uncertainty, while providing informative feedback to the user. Finally, to support users in responding to property violations, we must develop comprehensive methodological support for choosing between response strategies and guiding the evolution of SPLDCs.

Acknowledgements

We thank Michał Antkiewicz for his help with Alloy.

References

[1] P. C. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[2] K. Pohl, G. Boeckle, and F. van der Linden, *Software Product Line Engineering : Foundations, Principles, and Techniques*. Springer, 2005.

[3] H. Gomaa, *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison-Wesley, 2004.

[4] D. Benavides, S. Segura, and A. Ruiz-Corts, "Automated Analysis of Feature Models 20 years later: A Literature Review," *Information Systems*, vol. 35, no. 6, pp. 615 – 636, 2010.

[5] K. Czarnecki, S. Helsen, and U. Eisenecker, "Staged Configuration Using Feature Models," in *Proc. of SPLC'04*, ser. LCSE, vol. 3154, 2004, pp. 266–283.

[6] R. Salay, M. Famelis, J. Rubin, A. D. Sandro, and M. Chechik, "Lifting Model Transformations to Product Lines," in *Proc. of ICSE'14*, 2014, pp. 117–128.

[7] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A Classification and Survey of Analysis Strategies for Software Product Lines," *ACM Comput. Surv.*, vol. 47, no. 1, pp. 6:1–6:45, Jun. 2014.

[8] A. Ramirez, A. Jensen, and B. Cheng, "A Taxonomy of Uncertainty for Dynamically Adaptive Systems," in *Proc. of SEAMS'12*, 2012, pp. 99–108.

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

[10] N. Esfahani, S. Malek, and K. Razavi, "GuideArch: Guiding the Exploration of Architectural Solution Space Under Uncertainty," in *Proc. of ICSE'13*, 2013, pp. 43–52.

[11] A. Egyed, E. Letier, and A. Finkelstein, "Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models," in *Proc. of ASE'08*, 2008, pp. 99–108.

[12] M. Sabetzadeh, S. Nejati, M. Chechik, and S. Easterbrook, "Reasoning about Consistency in Model Merging," in *Proc. of LWI'10*, 2010.

[13] M. Famelis and M. Chechik, "Managing Design-Time Uncertainty," *Software & Systems Modeling*, pp. 1–36, 2017.

[14] K. Larsen and B. Thomsen, "A Modal Process Logic," in *Proc. of LICS'88*, 1988, pp. 203–210.

[15] M. Famelis, R. Salay, and M. Chechik, "Partial Models: Towards Modeling and Reasoning with Uncertainty," in *Proc. of ICSE'12*, 2012, pp. 573–583.

[16] M. Famelis, R. Salay, A. Di Sandro, and M. Chechik, "Transformation of Models Containing Uncertainty," in *Proc. of MODELS'13*, ser. LNCS, vol. 8107, 2013, pp. 673–689.

[17] R. Salay, M. Famelis, and M. Chechik, "Language Independent Refinement Using Partial Modeling," in *Proc. of FASE'12*, ser. LNCS, vol. 7212, 2012, pp. 224–239.

[18] A. Metzger and K. Pohl, "Software Product Line Engineering and Variability Management: Achievements and Challenges," in *Proc. of FOSE'14*, 2014, pp. 70–84.

[19] S. Neema, J. Sztipanovits, G. Karsai, and K. Butts, "Constraint-Based Design-Space Exploration and Model Synthesis," in *Proc. of EMSOFT'03*, 2003, pp. 290–305.

[20] A. Mjeda, A. Wasala, and G. Botterweck, "Decision Spaces in Product Lines, Decision Analysis, and Design Exploration: An Interdisciplinary Exploratory Study," in *Proc. of VaMoS'17*, 2017, pp. 68–75.

[21] J. van Gurp, J. Bosch, and M. Svahnberg, "On the Notion of Variability in Software Product Lines," in *Proc. of WISCA'01*, 2001, pp. 45–54.

[22] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, J. H. Obbink, and K. Pohl, "Variability Issues in Software Product Lines," in *Proc. of Workshop on Software Product-Family Engineering*, 2002, pp. 13–21.

[23] N. Esfahani and S. Malek, "Uncertainty in Self-Adaptive Software Systems." in *Proc. of Dagstuhl Seminar on Software Engineering for Self-Adaptive Systems*, R. de Lemos, H. Giese, H. Müller, and M. Shaw, Eds., 2012.

[24] N. Bencomo, P. Sawyer, G. S. Blair, and P. Grace, "Dynamically Adaptive Systems are Product Lines, Too: Using Model-Driven Techniques to Capture Dynamic Variability of Adaptive Systems." in *Proc. of SPLC'12*, 2008, pp. 23–32.

[25] I. Lytra, H. Eichelberger, H. Tran, G. Leyh, K. Schmid, and U. Zdun, "On the Interdependence and Integration of Variability and Architectural Decisions," in *Proc. of VaMoS'14*, 2013, pp. 19:1–19:8.

[26] J. Rubin and M. Chechik, "Quality of Merge-Refactorings for Product Lines," in *Proc. of FASE'13*, 2013, pp. 83–98.

[27] K. Czarnecki and M. Antkiewicz, "Mapping Features to Models: A Template Approach Based on Superimposed Variants," in *Proc. of GPCE'05*, 2005, pp. 422–437.

[28] K. Czarnecki and K. Pietroszek, "Verifying Feature-Based Model Templates against Well-Formedness OCL Constraints," in *Proc. GPCE'06*, 2006, pp. 211–220.

[29] S. Weaver, J. Franco, and J. Schlipf, "Extending Existential Quantification in Conjunctions of BDDs," *J. on Satisfiability, Boolean Modeling and Computation*, vol. 1, pp. 89–110, 2006.

[30] O. Djebbi, C. Salinesi, and G. Fanmuy, "Industry Survey of Product Lines Management Tools: Requirements, Qualities and Open Issues," in *Proc. of RE '07*, Oct 2007, pp. 301–306.

[31] M. Antkiewicz, K. Bak, A. Murashkin, R. Olaechea, J. H. J. Liang, and K. Czarnecki, "Clafer Tools for Product Line Engineering," in *Proc. of SPLC'13 Wrksp.*, 2013, pp. 130–135.

[32] B. Behringer, J. Palz, and T. Berger, "PEoPL: Projectional Editing of Product Lines," in *Proc. of ICSE'17 (to appear)*, 2017.

[33] M. Völter, J. Siegmund, T. Berger, and B. Kolb, "Towards User-Friendly Projectional Editors," in *Proc. of SLE'14*, vol. 8706, 2014, pp. 41–61.

[34] M. Völter, "Language and IDE Modularization and Composition with MPS," in *Proc. of GTTSE'11 Summer School*, 2013, pp. 383–430.

[35] D. Ratiu, M. Völter, Z. Molotnikov, and B. Schaetz, "Implementing Modular Domain Specific Languages and Analyses," in *Proc. of MoDeVVa'12*, 2012, pp. 35–40.

[36] M. Famelis, N. Ben-David, A. D. Sandro, R. Salay, and M. Chechik, "MU-MMINT: An IDE for Model Uncertainty," in *Proc. of ICSE'15 (Vol. 2)*, May 2015, pp. 697–700.

[37] A. Di Sandro, R. Salay, M. Famelis, S. Kokaly, and M. Chechik, "MMINT: A Graphical Tool for Interactive Model Management," in *Proc. of MODELS'15 Posters and Demos*, 2015, pp. 16–19.

[38] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2009.

[39] M. Antkiewicz and K. Czarnecki, "FeaturePlugin: Feature Modeling Plug-in for Eclipse," in *Proc. of OOPSLA'04 Workshop on Eclipse Technology eXchange*, 2004, pp. 67–72.

[40] T. Thum, C. Kaestner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich, "FeatureIDE: An Extensible Framework for Feature-Oriented Software Development," *Science of Computer Programming*, vol. 79, pp. 70 – 85, 2014.

[41] K. Czarnecki and K. Pietroszek, "Verifying Feature-based Model Templates against Well-Formedness OCL Constraints," in *Proc. of GPCE'06*, 2006, pp. 211–220.

[42] S. Ben-David, B. Sterin, J. M. Atlee, and S. Beidu, "Symbolic Model Checking of Product-Line Requirements Using SAT-based Methods," in *Proc. of ICSE'15*, 2015, pp. 189–199.

[43] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, S. Apel, and G. Saake, "SPL Conqueror: Toward Optimization of Non-Functional Properties in Software Product Lines," *Software Quality Journal*, vol. 20, no. 3, pp. 487–517, 2012.

[44] E. Mikk, Y. Lakhnech, M. Siegel, and G. J. Holzmann, "Implementing Statecharts in PROMELA/SPIN," in *Proc. of 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*, 1998, pp. 90–101.

[45] Object Management Group, *Object Constraint Language OMG Available Specification Version 2.0*, 2006. [Online]. Available: http://www.omg.org/cgi-bin/doc?formal/2006-05-01

[46] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. MIT press, 2012.