Université de Montréal

Modelling and Reasoning with Software Product Lines with Design Choices

par

Navpreet Kaur

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures et postdoctorales

en vue de l'obtention du grade de

Maître ès sciences (M.Sc.)

en Informatique

mars 2019

Sommaire

Les gammes de produits logiciels (Software Product Lines)(SPLs) permettent de gérer la variabilité qui apparaît dans les familles de modèles logiciels connexes en raison des variations des besoins des clients. Durant la conception de leurs modifications, les ingénieurs doivent considérer plusieurs conceptions de SPLs alternatives. Cependant, sans informations complètes sur les exigences de qualité souhaitées pour le SPL final, les ingénieurs sont confrontés à une incertitude quant au choix de la conception appropriée. Les formalismes et techniques existants ne conviennent pas à la modélisation et au raisonnement sur l'espace à deux dimensions défini par la variabilité et les choix conceptuels. Nous proposons une approche pour modéliser l'incertitude de conception dans les SPLs et, pour analyser et comprendre l'impact des choix conceptuels sur la qualité des SPLs, exprimé comme des propriétés. Nous définissons formellement les Gammes de produits logiciels avec des choix conceptuels (SPLDCs)(Software Product Lines with Design Choices) et nous décrivons une procédure pour les analyser et fournir une rétroaction appropriée aux ingénieurs basée sur l'ordre partiel des catégories de propriétés de SPLDC. Nous illustrons l'applicabilité de notre approche en utilisant un exemple entirément élaboré qui montre le type de rétroactions nuancées nécessaire pour des analyses significatives des SPLs en présence de choix conceptuels. Pour évaluer l'évolutivité de notre approche, nous utilisons notre approche sur de nombreux SPLDC et enregistrer des temps d'exécution.

Keywords : Ingénierie de ligne de produit, modélisation, choix de conception, variabilité, incertitude

Summary

Software product lines (SPLs) allow managing the variability that arises in families of related software models due to varying customer needs. While designing changes to them, engineers need to consider many alternative SPL designs. However, without complete information about the desired quality requirements of the final SPL, engineers face uncertainty about how to make the appropriate design choices. Existing formalisms and techniques are not well suited to modelling and reasoning about the two dimensional space defined by variability and design choices. We propose an approach for modelling design uncertainty in SPLs and for analyzing and understanding the impact of design choices in the quality of SPLs, expressed as properties. We formally define Software Product Lines with Design Choices (SPLDCs) and outline a procedure for analyzing them and providing appropriate feedback to engineers, based on the partial order of SPLDC property categories. We illustrate the applicability of our approach using a fully worked out example, that shows the kind of nuanced feedback necessary for meaningful analysis of SPLs in the presence of design choices. To evaluate the scalability of our approach we use our approach over many SPLDCs and record runtimes.

Keywords: Product line engineering, modeling, Design choices, variability, uncertainty

Contents

List of Tables

List of Figures

List of Acronyms

API:	Application Programming Interface
CM:	Choice Model
DM:	Domain Model
DSL:	Domain Specific Language
EMF:	Eclipse Modeling Framework
FM:	Feature Model
FOL:	First Order Logic
MOF:	Meta Object Facility
OMG:	Object Management Group
QBF:	Quantified Boolean Formula
RQ:	Research Question
SAT Solver:	Satisfiability Solver
SPL:	Software Product Line
SPLDC:	Software Product Line with Design Choice
SPLE:	Software Product Line Engineering
SPLOT:	Software Product Lines Online Tools
UML:	Unified Modeling Language
WM:	Washing Machine

Remerciements

I would like to acknowledge my indebtedness and render my warmest thanks to my supervisor, Professor Michalis Famelis, who made this work possible. His friendly guidance, expert advice, and encouragement have been invaluable through all the stages of this work. I also thank him, for providing the Financial Support throughout this period. I have been extremely lucky to have a supervisor who cared so much about my work, and who responded to my questions and queries so promptly. I am thankful to all the Professors and research scholars of the GEODES research group for providing me some useful insights and suggestions about this work and helping me in different ways.

I would like to thank my parents, whose love and guidance are with me in whatever I pursue. They are the ultimate role models. I would like to acknowledge all the hard work they endured for my studies. I am thankful to my brother Arshdeep Singh and his fiance Simrat for all their emotional support and motivation. Furthermore, I wish to thank my caring and supportive friends Joginder Singh, Manvi Virk, Gurpreet Kaur, Khady Fall, Safaa Allamy, Jatinder Singh, and Parminder Kaur for providing an unending inspiration and support.

Chapter 1

Introduction

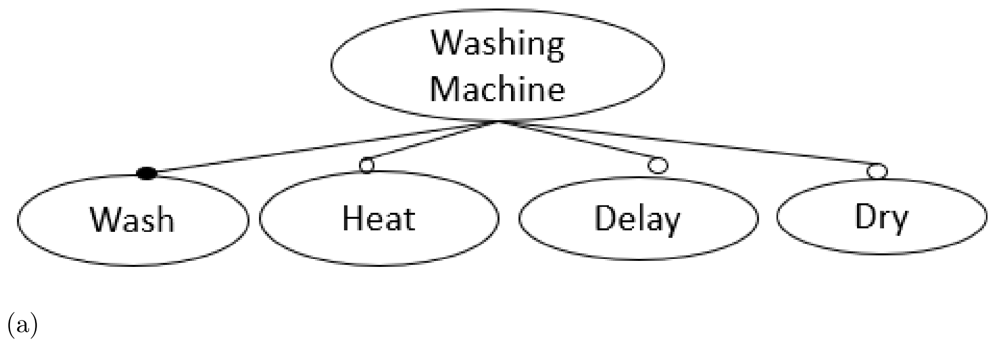
The way that industrial products are developed has changed significantly with time. Before the industrial revolution, products were crafted for individual customers. As the number of people able to afford different kinds of products increased, the production line was invented such as the Ford Production line. Production lines facilitate the production for a mass market rather than creating individualized products. This significantly reduced the production cost but at the same time reduced diversification possibilities. Customers were satisfied with standardized mass products for some time – but not everyone wanted the same kind of product, which increased demand to get individualized products. This led to the emergence of the *mass customization* approach, that is the large-scale development of products designated to individual needs. This meant increased technological expenditures, leading to costly products and lower profit margins for the producers. Therefore some companies started to use common platforms for their different kinds of products by anticipating in advance which parts will be used in different types of products. This approach enabled producers to offer a greater variety of products and at the same time reduced costs. The amalgamation of mass customization and a common platform allows both the reuse a common base of technology and, to bring out products in close accordance with individual needs [**28**].

In the case of software products, that already exhibit sprawling complexity, it makes things even more complex while implementing the fusion of the two approaches, as we need to manage the commonality and variability among different software variants. Therefore, there is a need to use some techniques dedicated to this scenario, known as *Software Product Line Techniques*. For instance, modern cars such as those made by General Motors, can contain tens of millions of lines of code, for implementing a variety of functionality such as powertrain control, safety features, climate control, and more. In addition to that, they have to deal with high variability to produce more than 60 models with additional variation to deal with the requirements differences in over 150 countries. To achieve this, they can make extensive use of software product line engineering techniques [**14**]. Combined with the need to raise the level of abstraction in software development, so that engineers can work using familiar notations, this lead to the emergence of model based techniques. Hence, different model-based techniques are also used by many companies [**14**].

1.1. Product lines

The need for high degrees of customizability and adaptability in software intensive systems compels developers to create, manage and maintain large families of similar but different product variants. To achieve this, they have at their disposal a wide range of available Software Product Line (SPL) practices that allow organizations to make long-term commitments to the maintenance of variability in sets of related products [**28**]. SPLs enables developers to take advantage of the common aspects of products related to each other with some predicted variability. In SPLs, variability is typically modeled using variation points, also known as features, and their inter-dependencies, typically expressed as feature models [**31, 28**]. Software product lines can significantly impact many aspects of product development such as: less development and maintenance costs, reduced time-to-market, improved product quality, improved customer satisfaction, reuse of artifacts and more [**32**]. SPLE techniques have been applied for many years in industries such as automotive, telecommunications, and others.

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 [**6**] We show the WM SPL in Figure 1.1. The WM SPL contains a feature model and a domain model. The feature model has four features: *Wash*, *Heat*, *Delay* and *Dry* that can be combined to generate several variants. *Wash* is a mandatory feature, that is, it must be present in all the valid feature configurations, however the other three features are optional. To generate a variant, the developers must select a valid subset of the features of WM. The product variant



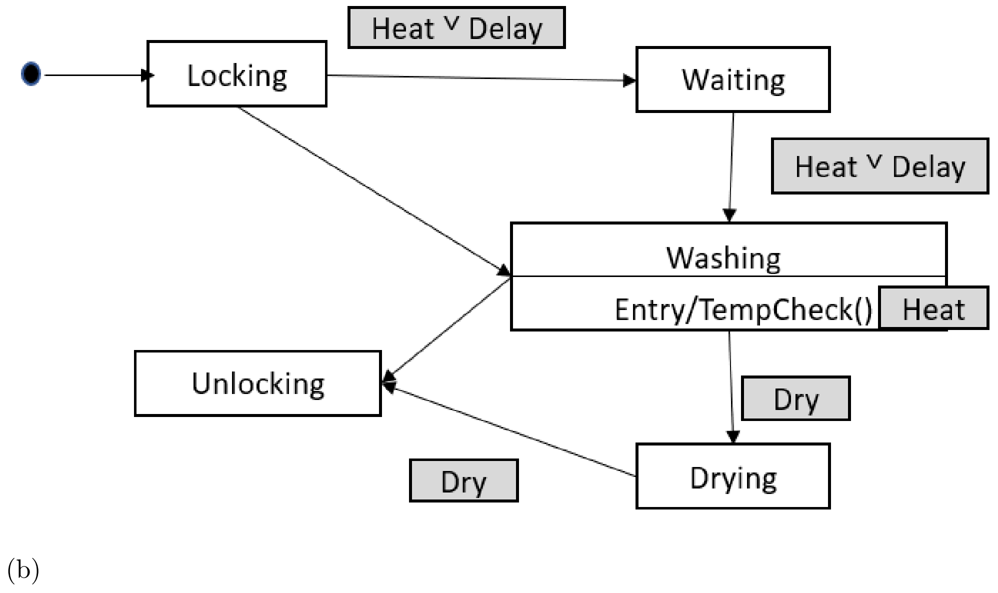


Figure 1.1. (a) Feature Model, (b) Domain Model of WM SPL

is then generated by appropriately evaluating the presence conditions of the elements of the domain model, shown in Figure 1.1(b) using grey boxes. For example, selecting features *Wash*, *Heat* and *Delay* will generate a variant of the washing machine example which is shown in Figure 1.2. These different combinations of features give different products to the user.

1.2. Design Uncertainty

SPL engineering allows the long-term maintenance of variability options by providing the ability to give individualized products to customers according to their need. However, engineers often need to express and reason about *short-term* design choices about products. These choices are a source of *design uncertainty* [**29**] and can result from many scenarios such as, dealing with different design alternatives, making decisions about product architecture,

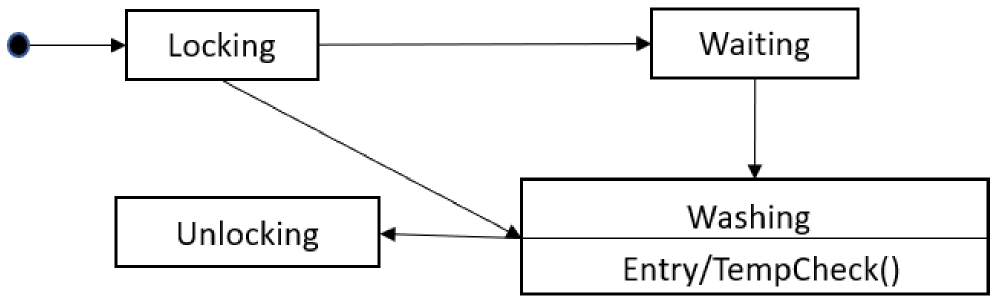


Figure 1.2. A Variant of WM SPL when features Heat and Delay are selected

resolving model inconsistencies, or resolving conflicting stakeholder requirements. There can be various design decisions about which developers can be uncertain. Such design decisions can impact various elements of an SPL definition. In our example, while designing the washing machine controller, the developers are not sure if the features *Heat* and *Delay* should be mutually exclusive or not. Also, the developers are uncertain if they should provide an option of heating incrementally or not. We represent these uncertainties as boolean design decisions. Each of the two design decisions is shown as a yellow annotation box with dashed outlines on the 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 Figure1.3(a), and **IncrementalHeat** as an annotation to a self-transition on the state Waiting, as shown in Figure1.3(b). By answering differently to these design decisions, developers can arrive to different SPL designs. For example, if they decide against **Mutex** and **IncrementalHeat**, the resulting SPL is the one shown in Figure 1.1.

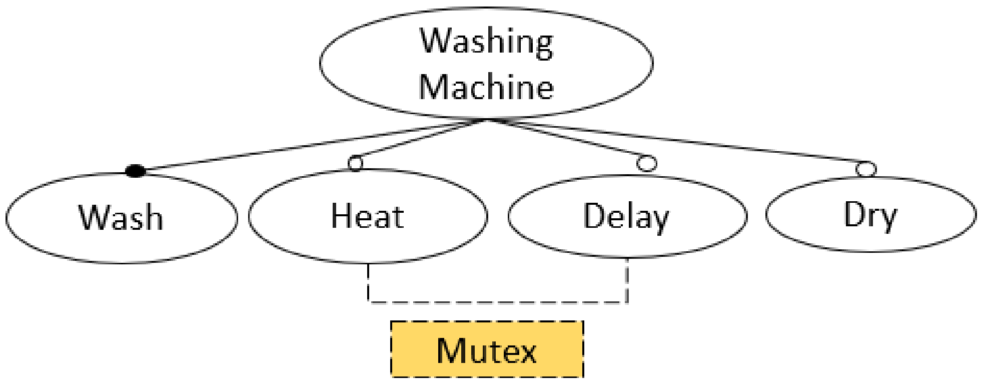
1.3. Problem Definition

The design uncertainty implies that there exists a space of possibilities, consisting of the set of product lines that can be created with different combinations of answers to the design choices. These design choices can therefore have significant impact on the quality requirements of the project.

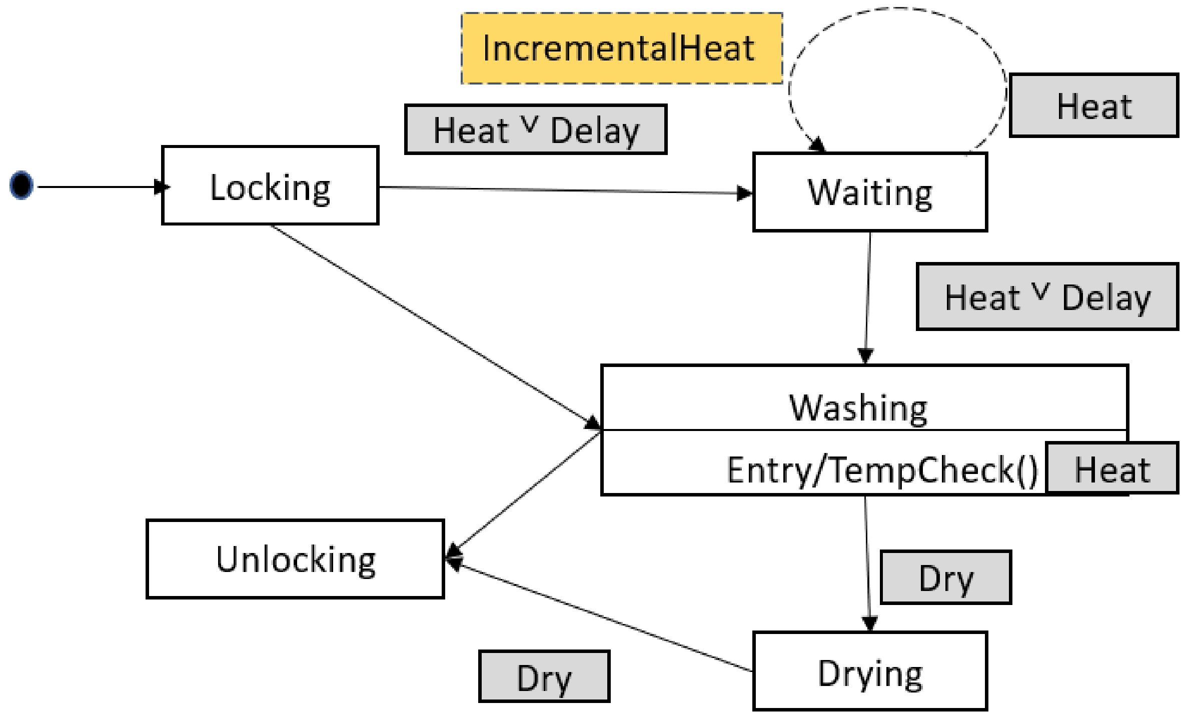
In previous work, it was shown that, instead of finalizing design decisions with little knowledge about their impact, developers can defer making these decisions until they acquire more information [**13**]. However, that work focused on single products rather than product families. If we consider design uncertainty in product lines then we get the combination of both long term variability commitments and short term choices of configurable software designs. We are therefore presented with challenges that cannot be adequately addressed by the state of the art. Suppose, the developers do not have enough knowledge to make the decisions yet, and they want to defer these decisions. But at the same time they want to check some properties of the system, such as consistency. For example, the developers can check the property R_2, shown in Table 2.1, if they want to make sure that whatever the decision they make in the future, and whatever configuration they choose, the state machine of the resulting controller product will always contain a final state. After analysis, the designers should know whether the system always has this property or not, regardless of the feature configuration they choose and design decisions they make in the future.

Checking these properties is a non trivial task, as both variability and uncertainty are present in the system and must therefore be taken into account in the check. For instance, in the example shown in Figure 1.3, we are uncertain whether the customer wants to have an option to heat for more than one cycle during a load or not. A developer may want to analyze the requirements models, even though they have some uncertainty, to better understand the impact of design decisions on quality. The variability of features and 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. However, existing techniques for variability management are not enough for reasoning in this scenario. In **[11]** Famelis et al. argued that managing the uncertainty induced by design choices is different from managing the variability induced by variation points. The goal of variability management is to support the different variants of a software product line in accordance to different needs among multiple customers. However, uncertainty management provides a means to explore and assess alternative designs, so that software developers can make informed decisions about the design choices [**16**].

While many approaches exist to manage and reason about variability in product lines [**33, 3**], and some techniques are developed for uncertainty management [**13**], there is no significant work to manage both dimensions. This work aims at dealing with the problem of expressing and reasoning about variability and design uncertainty in product lines simultaneously. In previous work [**11**] Famelis et al. outlined a research vision for design space exploration for SPLs. In this thesis, we expand on that work, using partial models, defined



(a)



(b)

Figure 1.3. (a) Feature Model, (b) Domain Model of WM SPL with design choices

in [**12**], to express decision points. A partial model is a representation of a set of models that could be obtained after resolving the design uncertainty. We call the systems having both decision points and variability points,such that the one in Figure 1.3, Software Product Line with Design Choices (SPLDCs) [**11**].

1.4. Research Contribution

The main contributions of this thesis are,

(1) the formal definition of product lines with design choices,

(2) a language for the formal specification of SPLDCs,

(3) a technique to analyze their quality requirements, expressed as structural properties, and

(4) a technique to generate feedback to engineers in the form of counterexample, in case the property is not satisfied for all the categories. We define the different levels of quality requirements of SPLDCs as categories of SPLDC-level properties in [**11**].

1.5. Thesis Structure

The rest of this document is organized as follows: We describe necessary background on SPL engineering in Chapter 2. In Chapter 3 we introduce the formalism for modelling SPLDCs and in Chapter 4 we describe how to analyze them, while generating appropriate feedback. In Chapter 5 we discuss implementation details of our approach. We discuss related work in Chapter 7 We evaluate the applicability and scalability of SPLDC analysis in Chapter 6 and conclude in Chapter 8.

Chapter 2

Background

In previous chapter, we explained the basics of product lines. In this chapter, we discuss some terminology related Software Product Line engineering using the *annotative* product line paradigm [**19**] and provide formal definition for each term. We also explain the analyzer we use for our approach.

2.1. Software Product Lines

This section provides a formal definition of software product lines.

Définition 2.1.1 (Software Product Lines)**.** *A Software Product Line (SPL) S is a 3-tuple* $< FM, DM, \mu >$ *consisting of: a Feature Model FM, a Domain Model DM, and feature mapping* μ *that maps features to the domain model entities.*

For instance, Figure 1.1 represents the SPL of a washing machine controller.

Définition 2.1.2 (Feature Model)**.** *A Feature Model* FM *is a graphical representation of 2-tuple,* $< F$,$\Phi_F >$ *where F is a set of features, and* Φ_F *is a formula representing variability constraints among them.*

There exist various model variants that provide different concrete syntaxes for representing constraints between features [**7**]. Our work is not dependent on any single feature model variant; instead we use them merely as a graphical representation of logical constraints [**5**]. Figure 1(a) represents the feature model of the WM example described above, where *Wash*, *Heat*, *Delay*, and *Dry* are features, with the constraints that *Wash* is mandatory which means that, *Wash* will be present in all valid configurations and other three features are optional.

Each feature uniquely corresponds to a propositional variable. In what follows, we use features and their corresponding variables interchangeably.

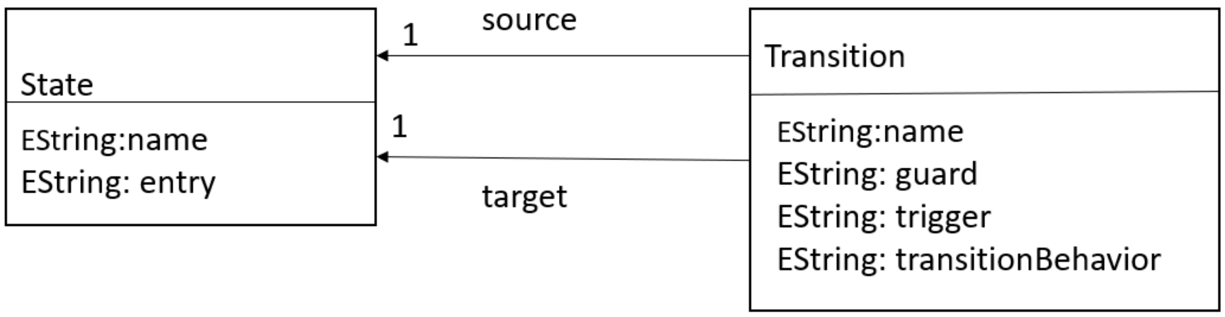


Figure 2.1. Simplified metamodel of state machines

Définition 2.1.3 (Domain Model)**.** *A Domain Model DM is a graphical representation of 2-tuple* $< D, \phi_D >$ *consisting of: a set of various model elements D, and a formula* ϕ_D *that represents the metamodel and well-formedness constraints.*

A Domain Model is a representation of entities in the system and relationships between them. The Domain Model can be described using a variety of modelling languages, such as state charts, class diagrams, etc. The domain models for the examples of this thesis are simplified versions of class diagram and state machines. Figure 1(b) depicts the domain model of the WM represented as a state chart. We show the simplified metamodel for state machines in Figure 2.1. For our example, we use state machines that conform to this metamodel. Hence, one of the well-formedness constraint for the domain models represented as state machines is that there is exactly one source for each transition, $\forall t : Transition, \exists s_1 : State \cdot (t = source = s_1) \wedge (\exists s_2 : State \cdot (t.source = s_2) \Rightarrow (s_1 = s_2))$. The formula ϕ_D represents the conjunction of all such constraints.

Définition 2.1.4 (Feature Mapping)**.** *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*. The formulas* ϕ_E *are known as* ***presence conditions****. If we encode the presence of an element e in a product by the propositional variable* v_e*, we can then logically express its feature mapping to the presence condition* ϕ_e *as* $\Phi_e = v_e \Leftrightarrow \phi_e$*.*

We represent presence conditions graphically using grey box annotations next to the graphical elements that they apply to. In the WM example, the state *Drying* has the presence condition ϕ_{Drying} =*Dry*. This means that state *Drying* is present in a product iff the feature *Dry* is selected.

2.2. Deriving Products

In this section we discuss the derivation of an individual product from the product line.

Définition 2.2.1 (Feature Configuration)**.** *A feature configuration ρ of a feature model $FM =< F, \Phi_F >$ is a subset of features from F that satisfies Φ_F. In other words, if in Φ_F we substitute every variable in ρ with* true *and all others with* false*, then the resulting expression evaluates to* true*. 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:

$$\{Wash, Dry\},$$

$$\{Wash, Heat\},$$

$$\{Wash, Delay, Dry\}$$

Définition 2.2.2 (Product Derivation)**.** *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 Figure 1.2, is a product derived from SPL using the feature configuration:

$$\rho = \{\mathit{Wash}, \mathit{Heat}, \mathit{Delay}\}$$

2.3. Levels of Property Satisfaction

While analyzing product lines, we can check its properties against different levels of satisfaction. In this section we discuss different levels of property satisfaction for a product line.

Définition 2.3.1 (Product Level Properties)**.** *A product level property R is a property that constrains a model without any variability, i.e., properties for individual products, rather than entire SPLs* [**11**]*.*

In this thesis, we focus on structural properties of models. For instance, in the WM example, the property R_2 "the model has a final state" is a product level property, which can

#	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

Table 2.1. Quality Requirements for WM example expressed as properties

be expressed[1] as:

$$R_2 = \exists s : State\ \nexists t : Transition \cdot (t.source = s)$$

The quality requirements for a product are expressed as a set R of product level properties which are desired to be true for that product. The quality requirements for the WM example are shown in Table 2.1.

Generally, we can define the degree of satisfaction of a quality requirement by the number of products of the SPL for which the corresponding property holds. Specifically, we identify two useful levels of satisfaction of a quality requirement:

Définition 2.3.2 (SPL levels of satisfaction)**.** *The level of satisfaction of a quality requirement expressed by a property R of an SPL S is:*

$\mathcal{A}$ **(All):** *if* $\forall p \in Conf(S) \cdot p \models R$.

$\mathcal{S}$ **(Some):** *if* $\exists p \in Conf(S) \cdot p \models R$.

In the WM example the quality requirement that a product always has exactly one final state (property R_2) is satisfied by all the products, so its level is $\mathcal{A}$. However, a requirement that no state has an entry action is clearly not satisfied by any feature that has the *Heat* feature, therefore, so its level is $\mathcal{S}$.

2.4. Alloy Analyzer

Given the requirements specifications for a system, we sometimes need to check if the property holds for all the models satisfying those requirements or not. This problem is equivalent to proving semantic entailment for first order logic (FOL) which is undecidable. So, rather than checking a property against all the possible models we can extract a relatively

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. More complex formalisations are of course possible.

small number of models from given requirements, and check that they satisfy the given property or not [**?**]. *Alloy* is a tool that implements this approach. Alloy is typically used for describing the structures and it is supported by the *Alloy Analyzer*, a tool for exploring those structures [**17**]. Alloy avoids the problem of the undecidability of FOL by putting a finite bound on the size of models, and check whether the property is satisfied by all the models whose size falls within the given bound and that satisfy all the requirements. We refer to this bound also as a "scope". A positive result to the check indicates that the property is valid for all models that come within the scope. The result is a proof only within the scope. There is no guarantee for larger models. However, a positive result provides us some confidence. To gain more confidence, we can increase the bound a little further. The negative response, on the other hand, is conclusive. It tells us that there exists at least one model that does not satisfy the property. Alloy is based on D. Jackson's small cope hypothesis, which states that negative answers tend to occur in small models already [**?**]. This allows us to assume that we can reach useful conclusions in small scopes, even if we cannot provide general proofs.

Chapter 3

Modelling Design Choices

In this chapter we describe the way in which design choices can be modelled in product lines.

3.1. SPLDCs

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. Famelis et al. introduced partial models as a formalism to represent design uncertainty [**15**], as well as different variants of partial models [**30**], that can express more complex types of partiality. In this thesis, we make two key assumptions:

(1) that modellers are aware of the design choices about the SPL that need to be addressed in the short term, and

(2) that for each such choice, a set of possible acceptable solutions has been elicited.

Design choices can therefore be represented as boolean choice variables.

Définition 3.1.1 (Design choice)**.** *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 *u*ncertainty 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.

To represent potential dependencies between design choices, we use a special feature model, called a "choice model":

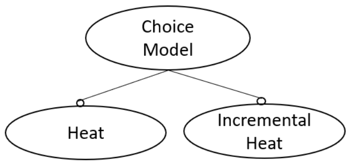


Figure 3.1. Choice Model for WM SPLDC

Définition 3.1.2 (Choice Model)**.** *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, drawn as a feature diagram, is shown in Figure 3.1. It consists of the two optional decisions **Mutex** and **IncrementalHeat**. Same as in Section 2.1, we do not assert any particular feature model dialect. We simply assume that the FM and CM are both expressed in the same dialect.

The design choices that capture the design uncertainty of modellers about an SPL, can then be mapped to the SPL elements as follows:

Définition 3.1.3 (Decision Mapping)**.** *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. If we encode the presence of an element s in an SPL design by the propositional variable v_s, we can then logically express its decision mapping to ϕ_s as $\Phi_s = v_s \Leftrightarrow \phi_s$.*

For instance, in WM shown in Figure 1.3(b) the transition looping over state *Waiting* is present in a product iff the choice **IncrementalHeat** is selected.

Using the above definitions, we can therefore formally define an SPLDC as:

Définition 3.1.4 (Software Product Line with Design Choices (SPLDC))**.** *An 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, Figure 1.3 represents an SPLDC of WM controller, where yellow boxes represent the design choices **Mutex** and **IncrementalHeat**.

The SPLDC allows representing both long term configuration options (in the FM of the underlying SPL definition) and short term design choices for which the developers have uncertainty (in the CM). If developers acquire information that allows them to resolve this uncertainty, they can make decisions about the design choices.

3.2. Deriving SPLs

Définition 3.2.1 (Design Decision)**.** *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**}}.

Given the previous definitions, the concretization of an SPLDCs is defined thus:

Définition 3.2.2 (Concretization)**.** *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. A* concretization *is a model without design uncertainty that results from resolving all design uncertainty in a partial model* [**15**]*. The set of all concretizations that can be derived from S is denoted by $Ch(S)$.*

For example the SPL shown in Figure 1.1 is a concretization of the WM SPLDC that can be derived using the decision $\alpha = \{\}$.

Chapter 4

Reasoning about SPLDCs

In this chapter we propose an approach to reason about the properties of SPLDCs. We discuss the various satisfaction levels for SPLDC quality requirements, expressed as properties. We provide an analysis procedure for analyzing the properties of SPLDCs and show its correctness.

4.1. Levels of SPLDC properties

Similar to the degree of satisfaction of a quality requirement by an SPL in Definition 2.3.2, we can define the degree of satisfaction of a quality requirement by an SPLDC, as the number of SPLs and number of products of an SPL, for which the corresponding property holds. Specifically, we identify four useful levels of satisfaction of a quality requirements:

Définition 4.1.1 (SPLDC levels of satisfaction)**.** *The level of satisfaction of a quality requirement expressed by a property R of an SPLDC SC is:*

$\mathcal{NA}$ **(Necessary All):** *if* $\forall s \in Ch(SC), \forall p \in Conf(s) \cdot p \models R$

$\mathcal{NS}$ **(Necessary Some):** *if* $\forall s \in Ch(SC), \exists p \in Conf(s) \cdot p \models R$

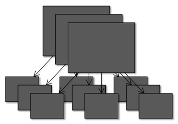
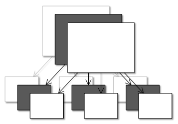
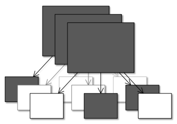
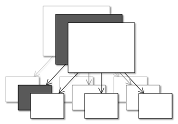
$\mathcal{PA}$ **(Possible All):** *if* $\exists s \in Ch(SC), \forall p \in Conf(s) \cdot p \models R$

$\mathcal{PS}$ **(Possible Some):** *if* $\exists s \in Ch(SC), \exists p \in Conf(s) \cdot p \models R$

For example, for the property R_2 given in Table 6.2 with respect to any SPLDC, the different levels 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.

Table 4.1. Levels of satisfaction of SPLDC quality requirements, expressed as properties.

	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 $\mathcal{F}$	$\forall\alpha \cdot \Phi^{\exists} \Rightarrow (\forall\rho \cdot \Phi \Rightarrow R)$	$\exists\alpha \cdot \Phi^{\exists} \wedge (\forall\rho \cdot \Phi \Rightarrow R)$	$\forall\alpha \cdot \Phi^{\exists} \Rightarrow (\exists\rho \cdot \Phi \wedge R)$	$\exists\alpha \cdot \Phi^{\exists} \wedge (\exists\rho \cdot \Phi \wedge R)$
Counterexample	α, ρ	ρ	α	–

$\mathcal{NS}$ **:** 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.

We summarize the four levels in Table 4.1.

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. In other words, if for example for an SPL S, a property R is satisfied at level $\mathcal{NA}$, it is also satisfied at levels, $\mathcal{PA}$, $\mathcal{NS}$, and $\mathcal{PS}$, while if it is satisfied at level $\mathcal{PA}$ or $\mathcal{NS}$ it is also satisfied at level $\mathcal{PS}$. Conversely, if the property is not satisfied for level $\mathcal{PS}$, it is not satisfied for any other level, etc. More formally, we observe that given the lemma of Bjorner [**2**] that a bounded poset of finite rank forms a lattice, the four SPLDC requirement satisfaction levels form the lattice:

In the following, we denote this lattice as $\mathcal{L}$.

$$\begin{tikzcd}
 & \mathcal{NA} \arrow[dl, dash] \arrow[dr, dash] & \\
\mathcal{NS} \arrow[dr, dash] & & \mathcal{PA} \arrow[dl, dash] \\
 & \mathcal{PS} &
\end{tikzcd}$$

4.2. Analysis procedure

Given this observation, we can define an analysis procedure for SPLDC properties, which we illustrate with the WM example. We assume as inputs the specification of an SPLDC K according to Definition 3.1.4, as well as a quality requirement R expressed as a first order logic property. Our aim is to understand the level of satisfaction of R in K.

First, we encode the SPLDC K in logic. Specifically, we construct the formula:

$$\Phi = \Phi_F \wedge \Phi_D \wedge \bigwedge_{e \in E} \Phi_e \wedge \Phi_C \wedge \bigwedge_{s \in S} \Phi_s$$

Where Φ_F is given in Definition 2.1.2, Φ_D in Definition 2.1.3, the set of formulas Φ_e over the set E of Domain Model elements in Definition 2.1.4, Φ_C in Definition 3.1.2, and the set of formulas Φ_s over the set S of elements of the SPL (features, domain model elements and feature mapping tuples) in Definition 3.1.3. 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 for it.

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, according to Table 4.1, using a satisfiability checker such as Sat4J [**21**]. The formulas $\mathcal{F}_{\mathcal{NA}}$,$\mathcal{F}_{\mathcal{PA}}$,$\mathcal{F}_{\mathcal{NS}}$, and $\mathcal{F}_{\mathcal{PS}}$ were first introduced in [**11**] but in this thesis we integrate them in a coherent modelling and analysis approach. In effect they *lift* product-level properties to the SPLDC level, allowing the quantification over design choices and features. We describe their construction below.

The formula Φ 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 $\Phi^{\exists}$ which encodes the dependency constraints on just the design choice variables; it is derived from Φ by *quantifying out* [**36**] all variables except for those representing design choices. This allows us to separate quantification over design choice and feature variables. 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 quality requirement is not satisfied at a particular level, we produce appropriate feedback by using the counterexample generated by the satisfiability checker 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, see Table 4.1. 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 2.1, 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 satisfiability checker. 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 2.1. 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 depends on the satisfiability 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 (obviously, as the domain model of WM, shown in Figure 1.3(b), does not contain any guards to begin with) and the solver does not need to generate a counterexample, as by definition every combination of α and ρ is a counterexample.

4.2.1. Correctness

Using the appropriate formalization formula $\mathcal{F}$ from Table 4.1, we can prove whether a property R of an SPLDC K is satisfied at the corresponding level, per Definition 4.1.1. Below, we state this as a theorem for level $\mathcal{NA}$. The theorems and proofs for levels $\mathcal{PA}$, $\mathcal{NS}$, and $\mathcal{PS}$ are analogous.

Théorème 4.2.1. *Given an SPLDC K, encoded logically as Φ, and a quality requirement R of K, expressed as a property, the level of satisfaction of R is $\mathcal{NA}$ iff the formula $\mathcal{F}_{\mathcal{NA}}$ (R)= $\forall\alpha \cdot \Phi^{\exists} \Rightarrow (\forall\rho \cdot \Phi \Rightarrow R)$ 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. □

4.3. Illustration

4.3.1. Setup

We illustrated our approach, to know about the kinds of insights we gain from the property analysis of SPLDCs. To do so, we transformed the counterexamples produced by *Alloy* for various property checks, to our domain specific language*Tyson* discussed in Chapter 5. The transformed counterexamples contain the feature configuration, choice configuration, or both, of the product or the product line that does not satisfy the property for a given level.

The type of feedback given for each level of property check is described in the last row of Table 4.1.

— If a property fails to satisfy level $\mathcal{NA}$, then it could be because of the feature configuration or the design choices that are made for the product for which the property fails. Therefore, we provide both feature configuration of the product and design decision of the product line containing that product for counterexample at level $\mathcal{NS}$.

— If the property fails at level $\mathcal{NS}$, then the possible reason for not satisfying the property is the design decision made for the SPL that fails to satisfy the property. So, for the counterexamples at level $\mathcal{NS}$ we give the design decision for the product line where the property is violated.

— For properties that do not satisfy level $\mathcal{PA}$, we provide the feature configuration for the product that violates the property.

— If the property is not satisfied for level $\mathcal{PS}$ then there is no product in SPLDC that satisfies the property. We do not provide any feedback for this level because the violation of the property is caused by the entities that are always present in the models. In other words, it is not caused because of a particular design decision or a feature configuration.

Based on the feedback we provide for each level, the developer can make sure not to make the design decisions or choose the feature configuration that does not satisfy the property. Developers can also add constraints to make sure that a particular feature configuration or design decision is never present in the system.

We use the **WM** example described in Chapter 1. The **WM** example is inspired from the motivating example in [**11**]. We used a slightly different version of the example in [**11**] to add

#	Property	Result	Counterexample Decision	Counterexample Configuration
1	∃ state S in DM: s is initial State.	SAT for all levels	-	-
2	∃ state S in DM: s is Final State.	SAT for all levels	-	-
3	∃ Transition T: t.source = t.target	Violates level $\mathcal{NS}$	IncrementalHeat	-
4	∃ Transition T: t.target=waiting	Violates level $\mathcal{PS}$	-	Wash, Dry

Table 4.2. Feedback generated for different property checks

another design choice to the example. The example is then represented in *Alloy* specification language [**17**], and the properties we want to analyze are written as Alloy Assertions.

4.3.2. Observations

We analyze the WM example against 4 properties. The feedback generated for each property check on **WM** is shown in Table 4.2. Two of these properties satisfy level $\mathcal{NA}$ which means that they satisfy all levels of property satisfaction. The property that checks there is no self loop in the state chart, fails at level $\mathcal{NS}$, so the feedback generated gives the decision **Incremental Heat** Fourth property checks if waiting is always present in system. The satisfaction level for this property is $\mathcal{PA}$, the feedback gives a feature configuration { Wash, Dry}

4.3.3. Discussion

The feedback we provide gives an insight to the developers about the issues present in the system. It gives the possible reason behind the violation of a property. To raise the satisfaction level for a property, the developers can make changes in accordance to the feedback provided. For instance, for Property #3 in the Table 4.2, we get the counterexample at level $\mathcal{NS}$, which gives contains the design decision **IncrementalHeat**. The result implies that the possible reason behind the non-satisfiability is the presence of decision **IncrementalHeat**.

So, to improve the level of satisfiability, the developer can decide against **IncrementalHeat** and check the property again.

Our study shows that the feedback generated for the property check, provides insights about the models that can help in improvement of a certain property satisfaction level. This gives us initial evidence for the usefulness and applicability of the study, giving us confidence to do further research on the topic.

We illustrated our approach on a small, synthetic example. However, despite its size, the example is enough to showcase that we can generate feedback that is useful in terms of helping modellers plan their responses to property violations. Furthermore, we are currently planning a bigger case study from the domain of mixed-criticality cyber-physical systems.

Chapter 5

Implementing SPLDCs in Tyson

In this Chapter we discuss the implementation of the approach discussed in Chapter 4. We have developed a Domain Specific Language (DSL), called *Tyson*, to simultaneously express variability and design uncertainty. In the following, we use the WM example to illustrate how Tyson can be used to model SPLDCs. The full Tyson specification of the WM example can be found in Appendix B.1

5.1. Language Basics

In order to specify SPLDCs, we need a language that can represent different entities of SPLDCs, and mapping among them. To accomplish this, we developed a textual language called *Tyson*. Tyson is a textual language developed using *Xtext*, a framework for the creation of programming languages and DSLs [**10**].

Xtext facilitates the development of DSLs by allowing the creation of a full infrastructure stack that includes a parser, linker, type checker, compiler, and editor with syntax highlighting. Xtext automatically generates an ecore metamodel for the language based on the grammar we write. Ecore is the core eclipse modeling framework (EMF), which allows us to describe models and provide run time support for the models. Ecore is an efficient API for generically manipulating EMF objects [**8**].

Tyson is specifically designed to express SPLDCs. The metamodel for *Tyson* is shown in the Appendix A.1. We can declare 3 kinds of models in *Tyson*: 1) a feature model that represent all the features and constraints among them, 2) a choice model that represents all the design choices the developer is uncertain about and, constraints among them, 3) domain model of the system. Apart from representing all different entities of SPLDCs, we also need to represent relations among them. We represent these relations in terms of two kinds of mappings: 1) Feature Mappings, which represent how features impact the presence of domain model entities 2) Decision Mappings, which represent how the different design choices impact product line entities. The Listing 5.1 shows the definition of choice model in *Tyson* having two design choices as described in Chapter 1.

Listing 5.1. Choice Model Definition in Tyson

```
CM { Mutex;
IncrementalHeat;
}
```

Listing 5.2 shows the definition of Feature Model shown in Figure 1.3(a) having 4 features. We also define a constraint that *Wash* is a Mandatory feature. Tyson also allows the definition of multiplicity constraints about a feature group. Similarly, we can also define constraints about design choices as a part of choice model definition.

Listing 5.2. Feature Model Definition in Tyson

```
FM  { Wash;
 Delay;
 Dry;
 Heat;
 FMConst:[ Mandatory(Wash)]
 }
```

Listing 5.3 defines the domain model shown in Figure 1.3(b) in *Tyson* having 5 states and 7 Transitions: *T1, T2, T3, T4, T5, T6, T7.* In this example the domain model is a State Chart. *Tyson* currently supports DMs conforming to simplified State Chart and UML Class Diagram metamodels.

Listing 5.3. Domain Model Definition in Tyson

```
StateChart{
                    State Locking;
                    State Waiting;
```

```
                State Washing;
                State Drying;
                State UnLocking;
                Transition T1: Locking to Waiting

                Transition T2: Waiting to Washing
                Transition T3: Locking to Washing
                Transition T4: Washing to UnLocking
                Transition T5: Washing to Drying
                Transition T6: Drying to UnLocking
                Transition T7: Waiting to Waiting
}
```

Mappings among various entities of SPLDC are shown in Listing 5.4. *F1*, *F2*, and *F3* are 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. *D1* and *D2* are Decision Mappings, that represents 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.

Listing 5.4. Mappings among SPLDC entities in Tyson

```
Mappings {
        FMap {
                F1:  {(Wash IN ) =>
                 AND  (Transition  T3 IN , Transition  T4 IN )}
                F2: {OR  (Heat IN , Delay IN ) =>
                 AND (Transition  T1 IN , Transition  T2 IN )}
                F3: {(Dry 2IN ) =>
                AND (Transition  T5 IN , Transition  T6 IN )}}
        DMap {
                D1: {Mutex IN  =>
                XOR (Feature  Heat IN  , Feature  Delay IN )}
                D2: {IncrementalHeat IN  =>
                Transition  T7 IN }}
                }
```

5.2. Tyson Semantics

The purpose of representing SPLDCs is to analyze and reason about their various properties. Therefore, we need to provide semantics to *Tyson*, that can facilitate the analysis of SPLDCs. To achieve this task, the *Tyson* models are transformed into *Alloy*. *Alloy* is a a lightweight formal method which allows us to do a bounded reasoning of first order logic. Alloy uses the Kodkod model finder and can leverage a variety of *SAT solvers* used as block box reasoners.

These *SAT Solvers* allow us to analyze different properties of the system written as assertions. To analyze SPLDCs, *Tyson* models are transformed into *Alloy*. We used *Acceleo* templates to do this transformation. Acceleo is a pragmatic implementation of the OMG MOF Model to Text standard [**27**]. In other words, we used an *Acceleo* transformation to give semantics to *Tyson* via a translation to a formal language: Alloy. *Acceleo* takes the models represented in *Tyson* and convert them as text that corresponds to *Alloy Specifications* for the *Tyson models*. The full Acceleo translation is provided in Appendix D.1 The full Alloy semantics of the WM example is provided in Appendix C.1.

Tyson does not provide a property specification language; instead, SPLDC properties are written directly in *Alloy*. In Section 5.3, we show how such properties can be checked for the 4 satisfaction levels in $\mathcal{L}$.

The Listing 5.5 represents the choice model definition in *Alloy* for the choice model in Listing 5.1.

Listing 5.5. Choice Model Definition in Alloy

```
abstract sig Choice{}
sig Mutex, IncrementalHeat extends Choice{}
```

```
//ChoiceModel definition
abstract sig ChoiceModel{
choice : set Choice}
```

Using Acceleo, the Feature Model defined in listing 5.2, will be transformed to the code represented in Listing 5.6. The *Fact* written in the code represents the constraint about feature *Wash*.

Listing 5.6. Feature Model Definition in Alloy

```
abstract sig Feature{}
one sig Wash, Heat , Delay, Dry extends Feature{}

abstract sig FeatureModel{
        feature: some Feature
}

fact {
        all f: FeatureModel | Wash in f.feature
}
```

Listing 5.3 which defines the domain model in *Tyson*, is transformed as Listing 5.7 in *Alloy*. The metamodel well-formedness constraint states that if a transition is present in a Domain Model, then its source state and target states will also be present in the Domain Model.

Listing 5.7. Domain Model Definition in Alloy

```
abstract sig State{}
one sig Locking, Waiting, Washing, Drying, Unlocking extends State{}


abstract sig Transition{
        source: one State,
        target: one State
}
one sig T1, T2, T3, T4, T5, T6, T7 extends Transition{}
```

```
fact {(T1.source = Locking) and (T1.target = Waiting)}
fact {(T2.source = Waiting) and (T2.target = Washing) }
fact {(T3.source = Locking) and (T3.target = Washing)  }
fact {(T4.source = Washing) and (T4.target = Unlocking) }
fact {(T5.source = Washing) and (T5.target = Drying)}
fact {(T6.source = Drying) and (T6.target = Unlocking) }
fact {(T7.source = Waiting) and (T7.target = Waiting)}


// DOMAIN MODEL DEFINITION
// container for model elements, i.e., the domain model
sig DomainModel{
        transition: some Transition,
        state: some State
}
// Metamodel well-formedness constraints
fact{all d: DomainModel | all s : State | some t: Transition |
 (t in d.transition) and ((s in t.source) or (s in t.target )) ⟹
(s in d.state)
else (s not in d.state)}
```

Listing 5.4 defines Feature Mappings and Decision Mappings. The Feature Mappings are transformed as Listing 5.8. The first *Fact* corresponds to the Feature Mapping *F1*, the second *Fact* corresponds to *F2*, and the third *Fact* corresponds to the mapping *F3*.

Listing 5.8. Feature Mapping in Alloy

```
fact {
        all p: Product |
                        (Wash in p.config.feature)⟹
                        ((T3 in p.dm.transition) and (T4 in p.dm.transition))
                else ((T3 not in p.dm.transition) and (T4 not  in p.dm.transition)
                   ↪ )
}


fact {
        all p: Product |
                        (Dry in p.config.feature)⟹
                        ((T5 in p.dm.transition) and (T6 in p.dm.transition))
                else ((T5 not in p.dm.transition) and (T6 not  in p.dm.transition)
                   ↪ )
}

fact {
        all p: Product|
                (Heat in p.config.feature) or (Delay in p.config.feature) ⟹
                        ((T1 in p.dm.transition) and (T2 in p.dm.transition))
                else ((T1 not in p.dm.transition) and (T2 not  in p.dm.transition)
                   ↪ )

}
```

The Product Line can be defined as Listing 5.9 in *Alloy*. It defines that a product line contains some product, where each product contains a valid feature configuration, and a Domain Model derived by the feature configuration.

Listing 5.9. Product Line Definition in Alloy

```
//Product Definition
abstract sig Product{
dm: one DomainModel,
config: one FeatureModel}

//Well formedness rules
fact{all f: FeatureModel | f in Product.config}
fact{all d: DomainModel | d in Product.dm}

//Product line definition
abstract sig SPL{
product : some Product
}
fact {all p: Product | p in SPL.product}
```

The decision mappings defined in Listing 5.4 are transformed to Listing 5.10. The first *Fact* corresponds to the decision Mapping *D1* and the second *Fact* corresponds to decision Mapping *D2*.

Listing 5.10. Decision Mapping in Alloy

```
//decision Mapping
fact { all dec: DesignChoices | all s: dec.spl | all p: s.product |
                Mutex in dec.cm.choice  ⟹
                not  ((Heat in p.config.feature) and (Delay in p.config.feature))
}

fact {
         all dec: DesignChoices | all s: dec.spl | all p: s.product |
        (IncrementalHeat in dec.cm.choice)  and
        (Heat in p.config.feature) and
        (Delay in p.config.feature) ⟹
                T7 in p.dm.transition
}
```

Listing 5.11 gives the definition on an SPLDC. An SPLDC contains some Design Choices, where each design choice contains a Choice configuration and an SPL derived by selected choices.

Listing 5.11. 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}
```

Listing 5.12 defines various symmetry breaking constraints for SPLDCs. These constraints stop Alloy from generating du2plicate instances of the SPLDC entities.

Listing 5.12. Symmetry Braking Constraints in Alloy

```
//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}
```

5.3. 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. *Alloy* allows us to write them as First Order Logic formulas.

For each property we write four assertions, one for each level of satisfaction in $\mathcal{L}$. Listing 5.13 shows the property R_2 ("the domain model has exactly one final state") in *Alloy*. The *check* statements in the listing are used to run the assertions. As discussed in Chapter 2, first order logic reasoning is undecidable without a bound so, each assertion is associated with a bound called *scope*, i.e., an integer value (Int) that represents the search space for the SAT solver while checking the validity of an assertion. The search for an instance is conducted within a scope. By specifying the scope, we make the search space finite by putting bounds on the sizes of the sets assigned signatures. It is to be noted that the scope does not declare the number of elements in the signature. Instead, it gives the bit width of integers (including the sign bit) and all integers that can be expressed using this bit width are included implicitly in the type Int. All integer computations are performed within the given bit width, and if, for a given instance, an expression's evaluation would require a larger bit width to succeed without overflow, the instance will not be considered by the analysis [**17**]. Therefore, when we get result for a property check, we can say that the result is valid within the given scope. To include larger expressions in their analysis, one can increase the scope value. In this example, the scope is set to 10. It is important to note that scope value is not set by default ,and developers can set scope according to their need. We keep a small value of scope here because size of our example is small. However, we can not make an argument about the optimal value for scope with respect to size of example. Finding optimal value for scope is a future work. The *Tyson* implementation allows declaring product-level properties as raw *String*s 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 $\mathcal{L}$.

Listing 5.13. SPLDC level properties in Alloy

```
//oneFinalNA
assert oneFinalNA {all pl: SPL | all p: pl.product| one d: p.dm | one s: d.state |
(s not in p.dm.transition.source)}

//oneFinalNS
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)}

//oneFinalPA
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)}

//oneFinalPS
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)}

check oneFinalNA for 10
check oneFinalNS for 10
check oneFinalPA for 10
check oneFinalPS for 10
```

Chapter 6

Evaluation

We aim to evaluate the scalability of our approach for analyzing the quality requirements of SPLDCs. We thus pose the following research question:

RQ (scalability) : How is the run time of the various property checks needed for SPLDC analysis affected by the size of the SPLDC?

In the scalability study, we applied the approach to SPLDCs of varying sizes, and collected the run times for each check. This allowed us to answer the RQ.

6.1. Experimental Setup

For the Scalability study, we applied the approach to randomly generated SPLDCs of varying sizes. 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, depending on the level of satisfaction for a property. The properties we check for each model are inspired from [**35**].

To generate feature and choice models for our study, we used the *SPLOT feature model repository* [**23**]. Feature models in SPLOT are expressed in a custom language for expressing logic constraints. We transformed 30 SPLOT feature models to *Tyson* feature models. To get the choice models, we transformed another 30 SPLOT feature models (different from the used to generate *Tyson* feature models) *Tyson* choice models. The SPLOT repository only contains feature models. To generate complete SPLDC specifications, we also need domain

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 6.1. Number of elements in each category

#	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)

Table 6.2. Properties Checked for scalability analysis

models. For this, we used the class diagrams of the metamodels from the AtlanMod Metamodel Zoo [**18**][1]

According to their size, we categorize feature models, choice models and domain models in three categories: small, medium and large. To generate the complete SPLDCs, we combine feature models, choice models and domain models from all three categories to generate 270 different SPLDCs, belonging to 27 different categories. Table 6.1 shows the number of elements in each category.

We generated random feature mappings and decision mappings for these SPLDCs. We then transformed all *Tyson* models to *Alloy* specifications.

We checked the three consistency properties shown in Table 6.2 for each model and recorded the run times to check the scalability of the approach. To study the effect of Alloy's *scope* in the analysis, we tried scopes of 20, 40, and 60. The results of property analysis were same regardless of scope, however the analysis became much slower. Below, we report results for scope 40. The machine we used to run these experiments is a LENOVO ThinkPad laptop, with 8GB RAM and Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz, 2701 Mhz, 2 Core(s), 4 Logical Processor(s).

1. `http://web.imt-atlantique.fr/x-info/atlanmod/index.php?title=Zoos`

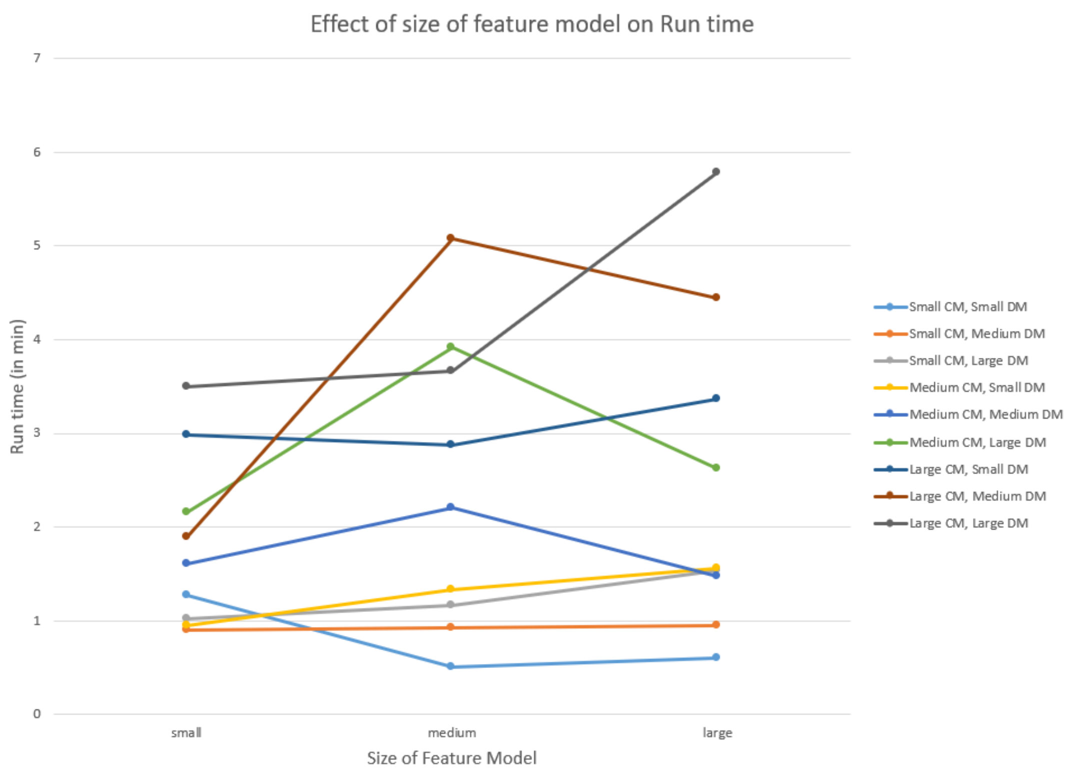


Figure 6.1. Effect of size of feature model on run time for different sizes of choice model and domain model

6.2. Results

To assess the scalability of the approach, we study how the run time for each check varies with the size of different elements of SPLDC. Figure 6.1, Figure 6.2, and Figure 6.3 shows the effects on the run time of the size of feature model, choice model and domain model respectively. The maximum time for a check was approximately 6 minutes for the category with large feature models, large choice models, and large domain models. Figure 6.1 shows a mixed trend in run time with respect to the size of feature models. On one hand, for some categories run time increases with increasing size of feature models. On the other hand, for some categories run time decreases despite an increase in the size of feature model.

However, figure 6.2 shows that in most of the cases, run time increases with increase in size of choice models.

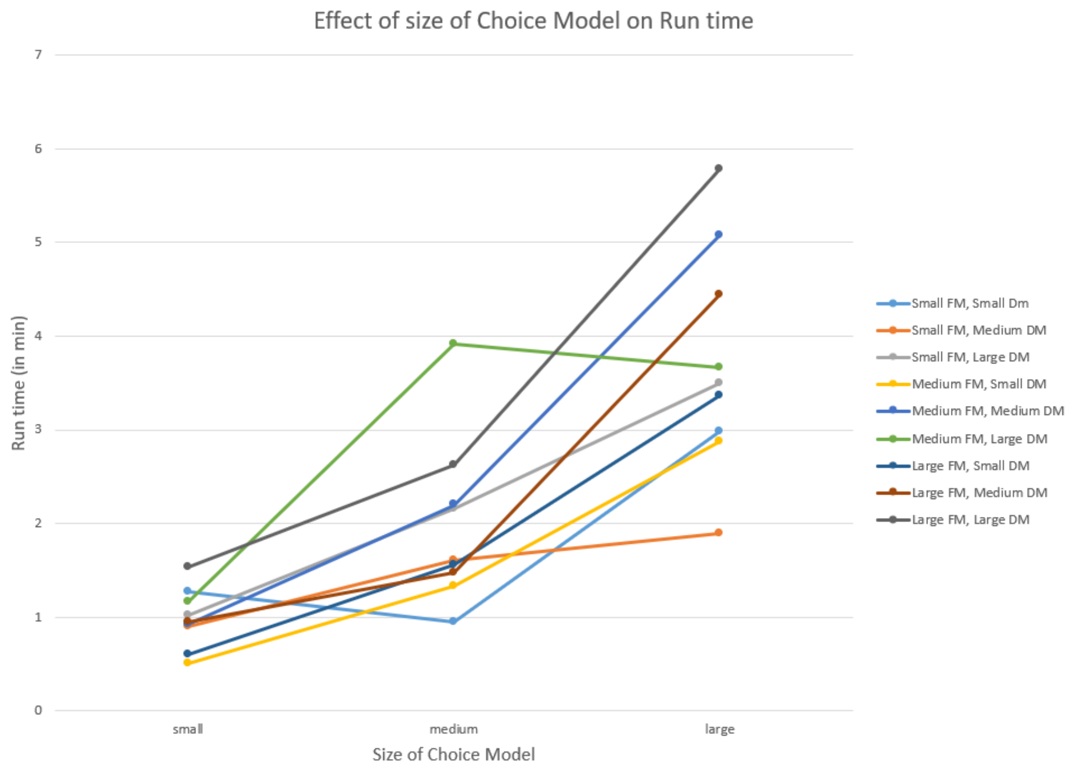


Figure 6.2. Effect of size of choice model on run time for different sizes of feature model and domain model

Figure 6.3 shows that 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.

6.3. Discussion

According to the results for the scalability study, we can see that in general for large SPLDCs, the run time for each check can take a few minutes. Figure 6.2 shows that size of choice model clearly impacts the run time. So, we conclude that for SPLDCs with more uncertainty, run time is larger. Figure 6.4 shows that *Alloy* scope has significant impact on the run time. Higher value of scope means higher confidence over the results as Alloy then considers larger expressions while analyzing a property, but that also leads to an increase in the run time. So, there is a trade-off between run time and confidence over the results. However, for the examples that we studied, we get the same results even increasing the scope. Hence, by considering the small scope hypothesis explained in Chapter 2, we set scope to

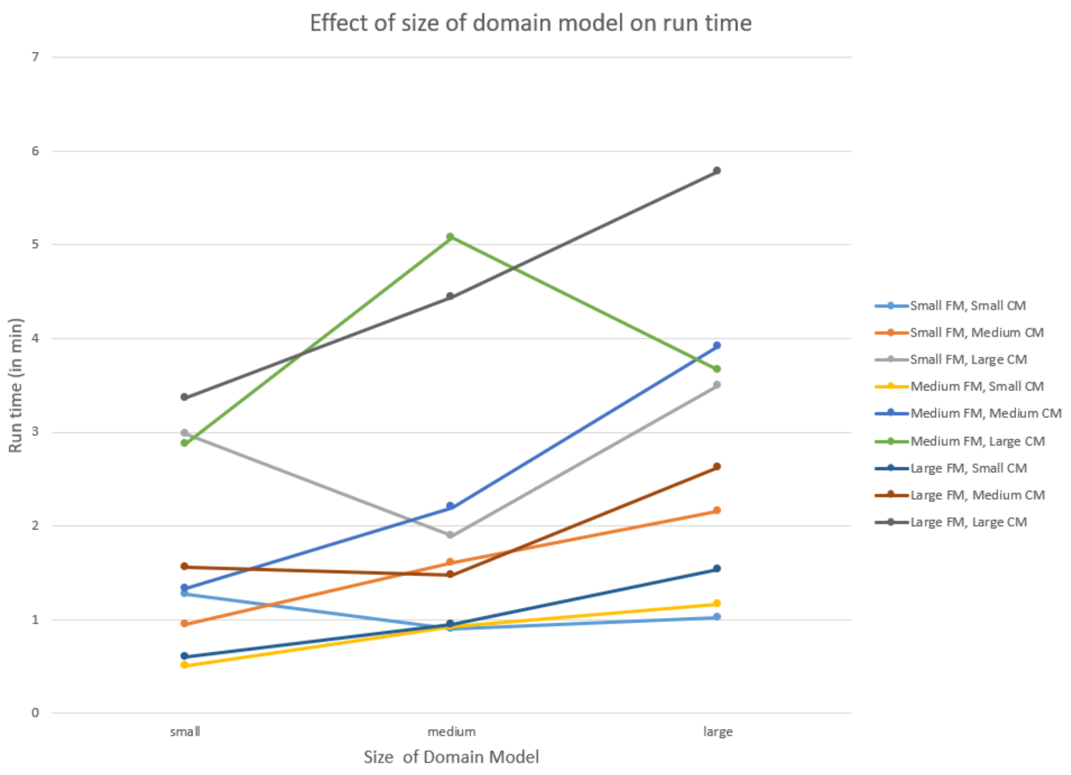


Figure 6.3. Effect of size of Domain model on run time for different sizes of choice model and Feature model

be 40 for our experiments. Overall, we note that with increasing the sizes of the different SPLDC components indicates some reasonable increases in run time. In other words, we do not see evidence that increasing the size of any of the components of an SPLDC can have a dramatic effect in the overall scalability of the approach. Even though the property checks took some minutes to complete, the overall cost did not outweigh the benefit of being able to reason in the presence of both variability and uncertainty in SPLDCs.

6.4. Threats to validity

Our evaluation is faced with various threats to the validity, discussed briefly below.

One threat to validity comes from our choice of experimental subjects. To mitigate the lack of real examples, we randomly generated SPLDCs. For this, we used real, publicly available models from SPLOT and the AtlanMod Metamodel Zoo, and only randomly generated the mappings. This allowed us to generate SPLDCs of various sizes, allowing us to explore the effect of the change in size. To mitigate the randomness, we used real

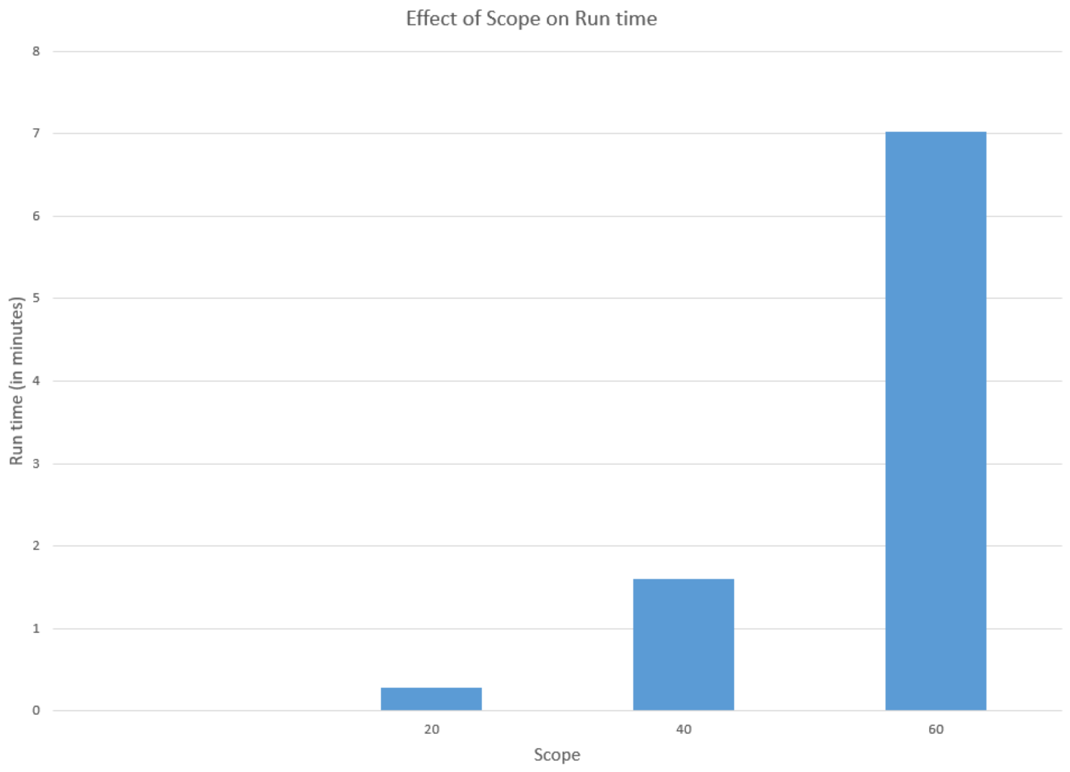


Figure 6.4. Effect of scope on run time

components (domain, feature and choice models) to generate SPLDCs. We are therefore confident that the resulting SPLDCs are *realistic*.

A second 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 effect, 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.

In the future we intend to experiment with even larger examples, as well as with other reasoning engines such as Clafer [**37**], and Alloy* [**26**] and alternative automated reasoning formalisms, such as QBF solving [**20**] and Answer Set Programming [**22**], with the aim to improve the efficiency of the implementation.

Chapter 7

Related Work

The work presented here follows up on a previously published vision paper [**11**], 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 [**24**] 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 [**15**].

Tran and Massacci [**34**] 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 [**9**] 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.

There exist two notable examples of research that attempts to explicitly model different kinds of choices present in SPL engineering. Lytra and colleagues [**16**] 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 and colleagues [**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, we note that 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 necessarily generating all products of an SPL [**33**], for instance, by checking feature-oriented analysis or exploitation of the variability information while analyzing.

Examples of the latter include techniques for model checking product lines [**4**], for automatically lifting analyses to account for variability [**25**], and for formalizing and analyzing behavioural specifications of product line requirements [**38**]. 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 (i.e., in the same feature model). However, as we have also illustrated in Chapter 6, it is useful to separate the two concerns, as design choices may very well affect the variability abstractions themselves. Moreover, unless we take care to model the two concerns separately, we can not easily express properties that quantify differently the two sets of choice variables and can thus generate nuanced feedback.

Chapter 8

Conclusion

8.1. Summary

We have described a formal approach for modelling and reasoning about design uncertainty in Software Product Lines (SPLs). We have shown how to formally represent design choices in different elements of an SPL definition using a formalism called "Software Product Lines with Design Choices" (SPLDCs). An SPLDC represents a two dimensional space defined by two axes: variability configurations and design decisions. Making design decisions on an SPLDC yields a concrete SPL design, which can then be configured to produce individual products by choosing among the different feature configurations. In order to analyze the quality requirements, expressed as properties, of SPLDCs we have defined four levels of requirements satisfaction, which allow different quantification over design decisions and product configurations. We then introduced an analysis procedure that leverages the formal semantics of SPLDCs, as well as the relationships between the different levels of quality requirement satisfaction. We developed a textual language to express and reason about the SPLDCs. We have illustrated the applicability of our approach by showing that nuanced feedback can be generated for an SPLDC, and we have studied the efficiency of reasoning, finding that generating nuanced feedback comes at a computational cost.

8.2. Limitations

In this section, we discuss some of the limitations of the proposed approach. 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. Furthermore, 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. Finally, our approach is focused on reasoning only about the structural properties of the models, leaving the behavioral properties of SPLDCS as a topic for future research.

8.3. Future Work

The future work for this problem may include the following.

(1) To further evaluate our approach with a large case study from the domain of real-time reactive embedded systems.

(2) To further investigate the use of other reasoning engines such as Clafer [**37**], and Alloy* [**26**] and alternative automated reasoning formalisms, such as QBF solving [**20**] and Answer Set Programming [**22**], with the aim to improve the efficiency of our implementation.

(3) To expand the expressiveness of the decision model, to allow representing more complex design decisions, such as those involving numeric ranges. This would allow our approach to be used for automated design space exploration for SPLs.

(4) To expand our approach to the analysis of requirements expressed as behavioral properties. This would require adapting our reasoning procedure to support specialized reasoning engines such as model checkers.

Appendix A

Tyson Metamodel

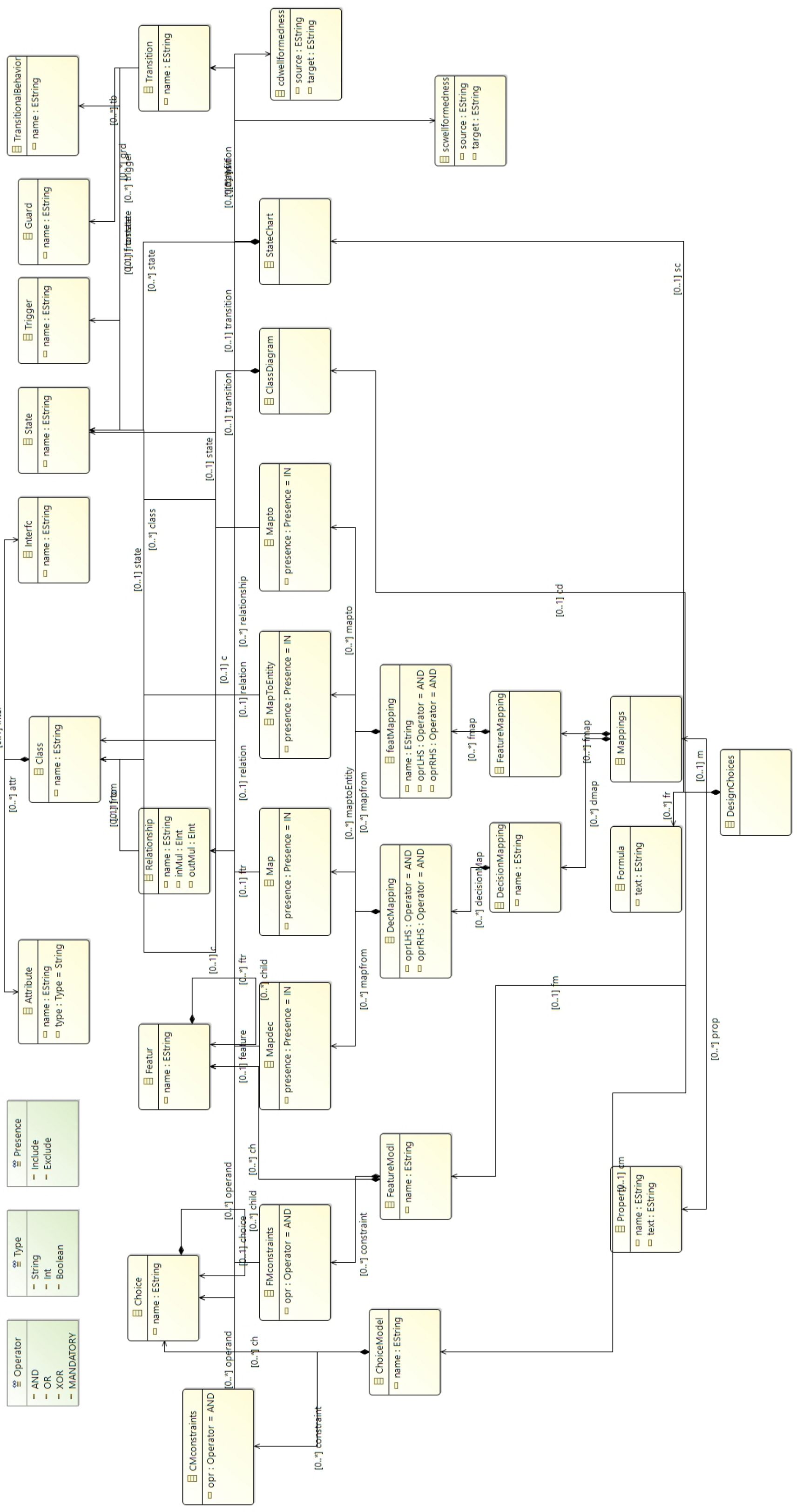


Figure A.1. Tyson Language metamodel

Appendix B

WM SPLDC in Tyson

Listing B.1. WM SPLDC definition in Tyson

```
CM { Mutex;
IncrementalHeat;
}

FM  { Wash;
 Delay;
 Dry;
 Heat;
 FMConst:[ Mandatory(Wash)]
 }

 StateChart{
                State Locking;
                State Waiting;
                State Washing;
                State Drying;
                State UnLocking;
                Transition T1: Locking to Waiting
                Transition T2: Waiting to Washing
                Transition T3: Locking to Washing

                Transition T4: Washing to UnLocking
                 Transition T5: Washing to Drying
                Transition T6: Drying to UnLocking
                Transition T7: Waiting to Waiting
}

Mappings {
        FMap {
                 F1:  {(Wash IN ) =>
                  AND  (Transition  T3 IN , Transition  T4 IN )}
                 F2: {OR  (Heat IN , Delay IN ) =>
                  AND (Transition  T1 IN , Transition  T2 IN )}
                 F3: {(Dry IN ) =>
                 AND (Transition  T5 IN , Transition  T6 IN )}}
        DMap {
                 D1: {Mutex IN  =>
                 XOR (Feature  Heat IN  , Feature  Delay IN )}
                 D2: {IncrementalHeat IN  =>
                 Transition  T7 IN }}
                 }
```

Appendix C

WM SPLDC in Alloy

Listing C.1. WM SPLDC in Alloy with some properties

```
// general definitions
abstract sig Feature{}
one sig Wash, Heat , Delay, Dry extends Feature{}

// FEATURE MODEL DEFINITION
abstract sig FeatureModel{
        feature: some Feature
}

// Wash is a necessary feature in all products
fact {
        all f: FeatureModel | Wash in f.feature
}

abstract sig State{}
one sig Locking, Waiting, Washing, Drying, Unlocking extends State{}

abstract sig Label{}
abstract sig Guard{}

abstract sig Transition{
        source: one State,
        target: one State,
        guard: lone Guard,
        label: lone Label
}
one sig Nospin extends Guard{}
one sig HeatEnabled_or_DelayEnabled , IncHeat, washstart, nospin, quickcool
    ↪ extends Label{}
one sig T1, T2, T3, T4, T5, T6, T7 extends Transition{}

fact {(T1.source = Locking) and (T1.target = Waiting) and (T1.label =
    ↪ HeatEnabled_or_DelayEnabled) and (T1.guard= none)}
fact {(T2.source = Waiting) and (T2.target = Washing) and (T2.label = washstart)
    ↪ and (T2.guard= none)}
fact {(T3.source = Locking) and (T3.target = Washing) and (T3.label = washstart)
    ↪ and (T3.guard= none) }
fact {(T4.source = Washing) and (T4.target = Unlocking) and (T4.label = quickcool)
    ↪  and (T4.guard= none)}
fact {(T5.source = Washing) and (T5.target = Drying)}
fact {(T6.source = Drying) and (T6.target = Unlocking) and (T6.label = quickcool)
    ↪ and (T6.guard= none)}
fact {(T7.source = Waiting) and (T7.target = Waiting) and (T7.label= IncHeat)and (
    ↪ T7.guard= none)}


// DOMAIN MODEL DEFINITION
// container for model elements, i.e., the domain model
sig DomainModel{
        transition: some Transition,
        state: some State
}
// Metamodel well-formedness constraints
fact{all d: DomainModel | all s : State | some t: Transition |
 (t in d.transition) and ((s in t.source) or (s in t.target )) ⟹
(s in d.state)
else (s not in d.state)}



//Product Definition
abstract sig Product{
```

```
dm: one DomainModel,
config: one FeatureModel}
one sig P1, P2, P3, P4, P5, P6, P7, P8, P9, P10, P11, P12 extends Product{}
//Well formdness rules
fact{all f: FeatureModel | f in Product.config}
fact{all d: DomainModel | d in Product.dm}

//Product line definition
abstract sig SPL{
product : some Product
}

fact {all p: Product | p in SPL.product}

fact {
        all p: Product |
                        (Wash in p.config.feature)⟹
                        ((T3 in p.dm.transition) and (T4 in p.dm.transition))
                else ((T3 not in p.dm.transition) and (T4 not  in p.dm.transition)
                   ↪ )
}

fact {
        all p: Product |
                        (Dry in p.config.feature)⟹
                        ((T5 in p.dm.transition) and (T6 in p.dm.transition))
                else ((T5 not in p.dm.transition) and (T6 not  in p.dm.transition)
                   ↪ )
}

fact {
        all p: Product|
                (Heat in p.config.feature) or (Delay in p.config.feature) ⟹
                        ((T1 in p.dm.transition) and (T2 in p.dm.transition))
                else ((T1 not in p.dm.transition) and (T2 not  in p.dm.transition)
                   ↪ )
```

```
}



//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}



//SPLDCs
//Choices
abstract sig Choice{}
sig Mutex, IncrementalHeat, Guards extends Choice{}

//ChoiceModel definition
abstract sig ChoiceModel{
choice : set Choice}

//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}
```

```
//decision Mapping
fact { all dec: DesignChoices | all s: dec.spl | all p: s.product |
                Mutex in dec.cm.choice  ⟹
                not  ((Heat in p.config.feature) and (Delay in p.config.feature))
}

fact {
         all dec: DesignChoices | all s: dec.spl | all p: s.product |
        (IncrementalHeat in dec.cm.choice)  and
        (Heat in p.config.feature) and
        (Delay in p.config.feature) ⟹
                T7 in p.dm.transition
}
one sig p1, p2, p3, p4, p5, p6 extends Product{}

//symmetry breaking constraints
fact{all c1, c2 : ChoiceModel | c1.choice = c2.choice ⟹
c1 = c2}
fact {all dc1, dc2 : DesignChoices | dc1.cm = dc2.cm ⟹
dc1=dc2}

//for all spl and for all products
assert initialState {all pl: SPL | all p: pl.product| some s: p.dm.state |
(s in p.dm.transition.source) and (s not in  p.dm.transition.target)}

assert initialStateone {all pl: SPL | all p: pl.product| one s: p.dm.state |
(s in p.dm.transition.source) and (s not in  p.dm.transition.target)}

assert initialStateoneNS {all pl: SPL | some p: pl.product| one s: p.dm.state |
(s in p.dm.transition.source) and (s not in  p.dm.transition.target)}

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

```
assert finalState {all pl: SPL | all p: pl.product| some s: p.dm.state |
(s not in p.dm.transition.source) and (s in  p.dm.transition.target)}

//oneFinalNA
assert oneFinalNA {all pl: SPL | all p: pl.product| one d: p.dm | one s: d.state |
(s not in p.dm.transition.source)}

//oneFinalNS
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)}

//oneFinalPA
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)}

//oneFinalPS
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)}

//no guards NA
assert NoGuardsNA{all pl: SPL|all p : pl.product |
 all t: p.dm.transition | t.guard =none}



//no guards NS
assert NoGuardsNS{all pl: SPL|some p : pl.product |
 all t: p.dm.transition | t.guard =none}

//no guards PA
assert NoGuardsPA{some pl: SPL|all p : pl.product |
 all t: p.dm.transition | t.guard =none}

//no guards PS
assert NoGuardsPS{some pl: SPL|some p : pl.product |
 all t: p.dm.transition | t.guard =none}

//incheatNA
```

```
assert incheatNA {all pl:SPL | all p: pl.product | T7 in p.dm.transition}

//incheatNS
assert incheatNs {all pl:SPL | some p: pl.product | T7 in p.dm.transition}

assert incheatPA {some pl:SPL | all p: pl.product | T7 in p.dm.transition}

assert incheatPS {some pl:SPL | some p: pl.product | T7 in p.dm.transition}

//waitPS
assert waitNs {all pl:SPL | some p: pl.product | T4 in p.dm.transition}



//p6
assert p6{some pl: SPL | some p: pl.product | some d: p.dm | T5 in d.transition }

check NoGuardsNA for 10
check NoGuardsNS for 10
check NoGuardsPA for 10
check NoGuardsPS for 10
check initialState for 10
check finalState for 10
check incheatNA for 10
check incheatNs for 10
check waitNs for 10
check initialStateone for 10
check initialStateoneNS for 10
check initialStateonePA for 10
check p6 for 10
check incheatPA for 10
check incheatPS for 10
```

Appendix D

Acceleo Template for Tyson to Alloy Transformation

Listing D.1. Acceleo template to transform Tyson models to Alloy

```
[comment encoding = UTF-8 /]
[module generate('http://www.xtext.org/example/mydsl/Tyson')]



[template public generateElement(dc : DesignChoices)]
[comment @main/]
[file ('/models/REAL-FM-3REAL-FM-18UEMLExtended.als', false, 'UTF
   ↪ -8')]
                    // BASIC DEFINITIONS
abstract sig Feature{
child: set Feature,
parent: lone Feature }
fact{all f : Feature | all c: f.child |
c.parent = f}

[for(fm:FeatureModl | dc.fm)]
[genfeatures(fm)/]
[genchildfeat(fm)/]
[for(fc :FMconstraints | fm.constraint)]
```

```
[getmandatoryfeat(fc)/]
[/for]
[/for]

// FEATURE MODEL DEFINITION
abstract sig FeatureModel{
        feature: some Feature
}

fact{all f: Feature |all fm: FeatureModel | f.child in fm.feature
   ↪ =>
f in fm.feature }

abstract sig Attribute{}
[for(a: ClassDiagram | dc.cd)]
[genattribute(a)/]
[/for]

abstract sig Class{
attr: set Attribute,
}

abstract sig Relationship{
from: lone Class,
to: lone Class,
inMul:lone Int,
outMul: lone Int}

[for(c: ClassDiagram | dc.cd)]
[genclasses(c)/]
```

```
[genclassattributes(c)/]
[genrelationships(c)/]
[genrelationdef(c)/]
[/for]


sig ClassDiagram{
class : some Class,
rel: some Relationship}


abstract sig Transition{}
abstract sig State{}
sig StateChart{
        transition: some Transition,
        state: some State
}


abstract sig DomainModel{
cd: lone ClassDiagram,
sc: lone StateChart
}


fact {
        all d: DomainModel | all c: d.cd | all r: Relationship | r
           ↪ in c.rel =>
        (r.from in c.class) and (r.to in c.class)
}
```

```
// Metamodel well-formedness constraints
//fact {
        //all d: DomainModel | all t: d.sc.transition | t in d.sc.
           ↪ transition =>
        //(t.source in d.sc.state) and (t.target in d.sc.state)
//}




//Product Definition
abstract sig Product{
dm: one DomainModel,
config: one FeatureModel}

//Well formdness rules
fact{all f: FeatureModel | f in Product.config}
fact{all d: DomainModel | d in Product.dm}

//Product line definition
abstract sig SPL{
product : some Product
}

fact {all p: Product | p in SPL.product}

//Feature Mapping
[for(m: Mappings | dc.m)]
```

```
[for(fm :FeatureMapping | m.fmap)]
[getfeatmap(fm)/]
[/for]
[/for]



fact {all f1, f2 : FeatureModel | f1.feature = f2.feature => f1=
    ↪ f2}
fact {all d1, d2 : StateChart | d1.transition = d2.transition =>
    ↪ d1=d2}
//fact {all t1, t2 : Transition | (t1.source = t2.source) and (t1.
    ↪ target = t2.target) =>
//t1=t2}
fact{all d1, d2: DomainModel | d1.sc=d2.sc=> d1=d2}
fact {all p1, p2 : Product | (p1.config = p2.config) and (p1.dm=p2
    ↪ .dm)=>
p1 = p2}
one sig P1, P2, P3, P4, P5, P6  extends Product{}



//SPLDCs
//Choices
abstract sig Choice{
child: set Choice,
parent: lone Choice}
fact{all f: Choice | all c: f.child |
c.parent = f }
[for(fm:ChoiceModel | dc.cm)]
```

```
[genchoices(fm)/]
[genchildch(fm)/]
[for(fc :CMconstraints | fm.constraint)]
[getmandatorych(fc)/]
[/for]
[/for]
```

```
//ChoiceModel definition
abstract sig ChoiceModel{
choice : set Choice}

fact{all c: Choice |all cm: ChoiceModel | c.child in cm.choice =>
c in cm.choice }


//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}

//decision Mapping
[for(m: Mappings | dc.m)]
[for(fm :DecisionMapping | m.dmap)]
[getdecmap(fm)/]
[/for]
[/for]




//symmetry breaking constraints
fact{all c1, c2 : ChoiceModel | c1.choice = c2.choice =>
c1 = c2}
fact {all dc1, dc2 : DesignChoices | dc1.cm = dc2.cm =>
dc21=dc2}




// There is no class related to itself
assert selfNA { all pl: SPL | all p: pl.product |
all r : p.dm.cd.rel | (r.to != r.from)
  }

assert selfNS { all pl: SPL | some p: pl.product |
all r : p.dm.cd.rel | (r.to != r.from)
```

```
  }

assert selfPA { some pl: SPL | all p: pl.product |
all r : p.dm.cd.rel | (r.to != r.from)
  }

assert selfPS { some pl: SPL | some p: pl.product |
all r : p.dm.cd.rel | (r.to != r.from)
  }


assert disconnectNA {all pl: SPL | all p: pl.product | no c: p.dm.
    ↪ cd.class |
(c not in p.dm.cd.rel.to) and (c not in p.dm.cd.rel.from)}

assert disconnectNS {all pl: SPL | some p: pl.product | no c: p.dm
    ↪ .cd.class |
(c not in p.dm.cd.rel.to) and (c not in p.dm.cd.rel.from)}

assert disconnectPA {some pl: SPL | all p: pl.product | no c: p.dm
    ↪ .cd.class |
(c not in p.dm.cd.rel.to) and (c not in p.dm.cd.rel.from)}


assert disconnectPS {some pl: SPL | some p: pl.product | no c: p.
    ↪ dm.cd.class |
(c not in p.dm.cd.rel.to) and (c not in p.dm.cd.rel.from)}
```

```
assert danglingNA{all pl: SPL | all p: pl.product | all r: p.dm.cd
    ↪ .rel|
(r.to !=none) and (r.from !=none)
}

assert danglingNS{all pl: SPL | some p: pl.product | all r: p.dm.
    ↪ cd.rel|
(r.to !=none) and (r.from !=none)
}

assert danglingPA{some pl: SPL | all p: pl.product | all r: p.dm.
    ↪ cd.rel|
(r.to !=none) and (r.from !=none)
}

assert danglingPS{some pl: SPL | some p: pl.product | all r: p.dm.
    ↪ cd.rel|
(r.to !=none) and (r.from !=none)
}

check selfNA for 20
check selfNS for 20
check selfPA for 20
check selfPS for 20

check disconnectNA for 20
check disconnectNS for 20
check disconnectPA for 20
check disconnectPS for 20
check danglingNA for 20
check danglingNS for 20
check danglingPA for 20
check danglingPS for 20

        [/file]

[/template]



[template public genfeatures(fm: FeatureModl)]
 one sig  [for (f: Featur | fm.eContents(Featur))  separator(',  ')
    ↪  ] [f.name/]  [/for] extends Feature{}
[/template]

[template public genchildfeat(fm: FeatureModl)]
[for(f: Featur | fm.ch)]
[if(f.child->size()<>0)]
 one sig  [for (f: Featur | f.child)  separator(',  ') ] [f.name/]
    ↪  [/for] extends Feature{}
fact{ [f.name/].child = [for(c: Featur | f.child) separator('+')]
   ↪ [c.name/] [/for] } [/if]
[/for]
[/template]



[template public genchoices(fm: ChoiceModel)]
 one sig  [for (f: Choice | fm.eContents(Choice))  separator(',  ')
    ↪  ] [f.name/]  [/for] extends Choice{}
[/template]
```

```
[template public genchildch(fm: ChoiceModel)]
[for(f: Choice | fm.ch)]
[if(f.child->size()<>0)]
one sig  [for (f: Choice | f.child)  separator(', ') ] [f.name/]
   ↪ [/for] extends Choice{}
fact{ [f.name/].child = [for(c: Choice | f.child) separator('+')]
   ↪ [c.name/] [/for] } [/if]
[/for]
[/template]


[template public genattribute(c: ClassDiagram)]
 one sig  [for (f: String | c.class.attr.name->asOrderedSet() )
    ↪ separator(', ') ] [f/] [/for] extends Attribute{}
[/template]


[template public genclasses(cd: ClassDiagram)]
 one sig  [for (f: Class | cd.eContents(Class))  separator(', ') ]
    ↪  [f.name/]  [/for] extends Class{}
[/template]


[template public genclassattributes(cd: ClassDiagram)]
[for(f: Class | cd.class)]
[if(f.attr->size()<>0)]
fact{ [f.name/].attr = [for(c: Attribute | f.attr) separator('+')]
   ↪  [c.name/] [/for] }
[/if]
```

```
[/for]
[/template]

[template public genrelationships(cd: ClassDiagram)]
 one sig  [for (f: Relationship | cd.eContents(Relationship))
    ↪ separator(', ') ] [f.name/]  [/for] extends Relationship{}
[/template]


[template public genrelationdef(cd: ClassDiagram)]
  [for (f: Relationship | cd.eContents(Relationship))]
fact{([f.name/].to = [f.to.name/]) and ([f.name/].from = [f.from.
   ↪ name/])
 and([f.name/].inMul = [f.inMul/]) and ([f.name/].outMul = [f.
    ↪ outMul/])}
[/for]
[/template]



[template public getmandatoryfeat(fc: FMconstraints)]
  [if(fc.opr= Operator::MANDATORY)]
fact{all fm: FeatureModel | [fc.operand.name/] in fm.feature}
[/if]

[if(fc.opr = Operator::AND)]
fact{all fm: FeatureModel | [fc.operand.name/].parent in fm.
   ↪ feature =>
([for(f:Featur | fc.operand.oclAsSet()) separator('and')] ([f.
   ↪ name/] in fm.feature) [/for]) }
[/if]
```

```
[if(fc.opr = Operator::OR)]
fact{all fm: FeatureModel | [fc.operand.name/].parent in fm.
   ↪ feature =>
([for(f:Featur | fc.operand.oclAsSet()) separator('or')] ([f.name
   ↪ /] in fm.feature) [/for] )}
[/if]

[/template]



[template public getmandatorych(fc: CMconstraints)]
  [if(fc.opr = Operator::MANDATORY)]
fact{all fm: ChoiceModel | [fc.operand.name/] in fm.choice}
[/if]

[if(fc.opr = Operator::AND)]
fact{all fm: ChoiceModel | [fc.operand.name/].parent in fm.choice
   ↪ =>
([for(f:Choice | fc.operand.oclAsSet()) separator('and')] ([f.
   ↪ name/] in fm.choice) [/for] )}
[/if]

[if(fc.opr = Operator::OR)]
fact{all fm: ChoiceModel | [fc.operand.name/].parent in fm.choice
   ↪ =>
([for(f:Choice | fc.operand.oclAsSet()) separator('or')] ([f.name
   ↪ /] in fm.choice) [/for]) }
[/if]
[/template]
```

```
[template public getfeatmap(fm: FeatureMapping)]
[for(fmap: featMapping | fm.fmap)]
fact{all p: Product |
[if(fmap.oprLHS= Operator::AND)][for(f: Map | fmap.mapfrom)
   ↪ separator('and')] ([f.ftr.name/] in p.config.feature) [/for
   ↪ ][/if]
[if(fmap.oprLHS=Operator::OR)][for(f: Map | fmap.mapfrom)
   ↪ separator('or')] ([f.ftr.name/] in p.config.feature) [/for
   ↪ ][/if]
=>
[if(fmap.oprRHS=Operator::AND)][for(f: Mapto | fmap.mapto)
   ↪ separator('and')] ([f.relation.name/] in p.dm.cd.rel) [/for
   ↪ ][/if]
[if(fmap.oprRHS=Operator::OR)][for(f: Mapto | fmap.mapto)
   ↪ separator('or')] ([f.relation.name/] in p.dm.cd.rel) [/for
   ↪ ][/if]
}[/for]
[/template]


[template public getdecmap(fm: DecisionMapping)]
[for(fmap: DecMapping | fm.decisionMap)]
fact{all dec: DesignChoices | all s: dec.spl | all p: s.product |
[if(fmap.oprLHS=Operator::AND)][for(f: Mapdec | fmap.mapfrom)
   ↪ separator('and')] ([f.choice.name/] in dec.cm.choice) [/for
   ↪ ][/if]
[if(fmap.oprLHS=Operator::OR)][for(f: Mapdec | fmap.mapfrom)
    ↪ separator('or')] ([f.choice.name/] in dec.cm.choice) [/for
    ↪ ][/if]
=>
[if((fmap.oprRHS=Operator::AND))][for(f: MapToEntity | fmap.
    ↪ maptoEntity) separator('and')] ([f.relation.name/] in p.dm.
    ↪ cd.rel )[/for][/if]
[if(fmap.oprRHS=Operator::OR)][for(f: MapToEntity | fmap.
    ↪ maptoEntity) separator('or')] ([f.relation.name/] in p.dm.cd
    ↪ .rel )[/for][/if]
}[/for]
[/template]
```

Bibliography

[1] Simon Barner, Alexander Diewald, Fernando Eizaguirre, Anatoly Vasilevskiy, and Franck Chauvel. Building product-lines of mixed-criticality systems. In *2016 Forum on Specification and Design Languages, FDL 2016, Bremen, Germany, September 14-16, 2016*, pages 1–8, 2016.

[2] Anders Björner, Paul H. Edelman, and Günter M Ziegler. Hyperplane arrangements with a lattice of regions. 1990.

[3] L. Chen, M. Ali Babar, and N. Ali. Varibility management in software product lines: a syatematic review. In *Proceeding SPLC '09 Proceedings of the 13th International Software Product Line Conference*, 2009.

[4] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 335–344, New York, NY, USA, 2010. ACM.

[5] K. Czarnecki and A. Wasowski. Feature diagrams and logics: There and back again. In *11th International Software Product Line Conference (SPLC 2007)*, pages 23–34, Sep. 2007.

[6] Krzysztof Czarnecki, Michal Antkiewicz, Chang Hwan Peter Kim, Sean Lau, and Krzysztof Pietroszek. Model-driven software product lines. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 126–127, New York, NY, USA, 2005. ACM.

[7] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[8] Inc. Eclipse Foundation. Eclipse modeling framework.

[9] Naeem Esfahani and Sam Malek. Uncertainty in Self-Adaptive Software Systems. In R. de Lemos, H. Giese, H. Müller, and M. Shaw, editors, *Software Engineering for Self-Adaptive Systems 2, Lecture Notes in Computer Science Hot Topics*. 2012.

[10] M. Eysholdt and H. Behrens. Xtext:implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, 2010.

[11] M. Famelis, J. Rubin, Krzysztof, R. Salay, and M. Chechik. Software Product Lines with Design Choices: Reasoning about Variability and Design Uncertainty. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 93–100. IEEE, 2017.

[12] M. Famelis, R. Salay, and M. Chechik. Partial models: Towards modeling and reasoning with uncertainty. In *2012 34th International Conference on Software Engineering (ICSE)*, 2012.

[13] Michalis Famelis and Marsha Chechik. Managing design-time uncertainty. *Software & Systems Modeling*, March 2017.

[14] Michalis Famelis, Levi Lúcio, Gehan Selim, Alessio Di Sandro, Rick Salay, Marsha Chechik, James R. Cordy, Juergen Dingel, Hans Vangheluwe, and Ramesh S. Migrating automotive product lines: A case study. In Dimitris Kolovos and Manuel Wimmer, editors, *Theory and Practice of Model Transformations*, pages 82–97, Cham, 2015. Springer International Publishing.

[15] Michalis Famelis, Rick Salay, and Marsha Chechik. Partial Models: Towards Modeling and Reasoning with Uncertainty. In *Proc. of ICSE'12*, pages 573–583, 2012.

[16] 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 *Proceedings of Eighth International Workshop on Variability MOdelling of Software-Intensive Systems*, 2014.

[17] D. Jackson. alloy: a language and tool for relational models, 2012.

[18] Frédéric Jouault and Jean Bézivin. Km3: A dsl for metamodel specification, 2006.

[19] Christian Kästner and Sven Apel. Integrating Compositional and Annotative Approaches for Product Line Engineering. In *Proc. of GPCE'08*, pages 35–40, 2008.

[20] William Klieber. Formal verification using quantified boolean formulas (qbf), 2014.

[21] Daniel Le Berre and Anne Parrain. The SAT4J library, Release 2.2, System Description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.

[22] Vladimir Lifschitz. What is answer set programming?, 2008.

[23] Marcilio Mendonca, Moises Branco, and Donald Cowan. Splot: software product lines online tools. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 761–762. ACM, 2009.

[24] Andreas Metzger and Klaus Pohl. Software Product Line Engineering and Variability Management: Achievements and Challenges. In *Proc. of FOSE'14*, pages 70–84, 2014.

[25] Jan Midtgaard, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej WÄ…sowski. Systematic derivation of correct variability-aware program analyses. *Science of Computer Programming*, 105:145 – 170, 2015.

[26] Aleksandar Milicevic, Joseph P. Near, Eunsuk Kang, and Daniel Jackson. Alloy*: a general-purpose higher-order relational constraint solver. *Formal Methods in System Design*, Jan 2017.

[27] Obeo Network. Acceleo, 2017.

[28] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, Berlin, Heidelberg, 2005.

[29] J. Ramirez, C. Jensen, and HC. Cheng. A taxonomy of uncertainty for dynamically adaptive systems. In *Proceedings of 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2012.

[30] Rick Salay, Michalis Famelis, and Marsha Chechik. Language independent refinement using partial modeling. In Juan de Lara and Andrea Zisman, editors, *Fundamental Approaches to Software Engineering*, pages 224–239, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[31] P. Y. Schobbens, P. Heymans, and J. C. Trigaux. Feature diagrams: A survey and a formal semantics. In *14th IEEE International Requirements Engineering Conference (RE'06)*, pages 139–148, Sept 2006.

[32] Yuri Sprockel. The impact of software product lines from a product management perspective, 2013.

[33] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.*, 47(1):6:1–6:45, June 2014.

[34] Le Minh Sang Tran and Fabio Massacci. An approach for decision support on the uncertainty in feature model evolution. In *2014 IEEE 22nd International Requirements Engineering Conference*, 2014.

[35] Ragnhild Van Der Straeten, Tom Mens, Jocelyn Simmonds, and Viviane Jonckers. Using description logic to maintain consistency between uml models. pages 326–340, 10 2003.

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

[37] Markus Weckesser, Malte Lochau, Michael Ries, and Andy Schürr. Towards complete consistency checks of clafer models. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Feature-Oriented Software Development*, FOSD 2017, pages 11–20, New York, NY, USA, 2017. ACM.

[38] Jiale Zhou, Yue Lu, Kristina Lundqvist, Henrik Lönn, Daniel Karlsson, and Bo Liwang. Towards feature-oriented requirements validation for automotive systems. In *2014 IEEE 22nd International Requirements Engineering Conference*, 2014.