

THESIS

DESIGN AND EVALUATION OF THE FAMILIAR TOOL

Submitted by

Aleksandar Jaksic

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Spring 2014

Master's Committee:

Advisor: Robert B. France

Charles W. Anderson

Sudipto Ghosh

Lucy J. Troup

Copyright by Aleksandar Jaksic 2014

All Rights Reserved

ABSTRACT

DESIGN AND EVALUATION OF THE FAMILIAR TOOL

Software Product Line Engineering (SPLE) aims to efficiently produce multiple software products, on a large scale, that share a common set of core development features. Feature Modeling is a popular SPLE technique used to describe variability in a product family.

FAMILIAR (FeAture Model sCript Language for manIpulation and Automatic Reasoning) is a Domain-Specific Modeling Language (DSML) for manipulating Feature Models (FMs). One of the strengths of the FAMILIAR language is that it provides rich semantics for FM composition operators (aggregate, merge, insert) as well as decomposition operators (slice).

The main contribution of this thesis is to provide an integrated graphical modeling environment that significantly improves upon the initial FAMILIAR framework that was text-based and consisted of loosely coupled parts. As part of this thesis we designed and implemented a new FAMILIAR Tool that provides (1) a fast rendering framework for the graphically representing feature models, (2) a configuration editor and (3) persistence of feature models. Furthermore, we evaluated the usability of our new FAMILIAR Tool by performing a small experiment primarily focusing on assessing quality aspects of newly authored FMs as well as user effectiveness and efficiency.

ACKNOWLEDGEMENTS

This work would not be possible without help and support of many people.

First of all, I would like to thank my adviser, Dr. Robert B. France, for guiding my research in many ways, his continued support, and great inspiration. Dr. France introduced me to both research world as well as SPLE. He continues to nurture me in my academic endeavors. I am grateful to my committee members, Dr. Charles W. Anderson, Dr. Sudipto Ghosh, and Dr. Lucy J. Troup for their teaching and roles. Dr. Troup and also Dr. Jaime Ruiz provided a valuable help when I was planning the experiment.

Special thanks to Dr. Mathieu Acher. Although Mathieu managed to misspell my first name countless number of times, and occasionally ignored my emails for weeks (and still does both :), I owe him an enormous credit. This thesis would not be possible without Mathieu's friendly guidance, and his expertise in SPL, DSML, feature modeling, and, most importantly, contribution to the FAMILIAR project. Special thanks are also due to Dr. Philippe Collet and Simon Urli for conducting the FAMILIAR Tool experiment in their graduate SPL class. I am also very grateful to all CS graduate students who participated in the experiment: fourteen students from UNSA and three from CSU. Thanks to prof. Bruce Draper, graduate program director, and James L. Peterson, key academic advisor, who were always responsive, and provided answers to all of my questions, especially during my first graduate year at CSU. I am also grateful to Dr. Geri Georg, who gave me invaluable advice when I was looking for an advisor.

I would like to thank my employer Microsoft for setting a fine example as a world class software and services company that supports continued education and for encouraging my

research. In addition, I am indebted to my managers at work: Mitch Eatough and Vijayalakshmi Ramkumar for granting approvals for my graduate work and for taking a personal interest in it.

Finally, all my love goes to my wife Tamara, and children Aleksandra, Gorana and Jana.

Ви сѝе све шѝо имам у живоѝу, мој ѝонос, љубав и срећа! Само дани које сам ѝровео са Вама, у којима сам Вас заѝрллио и ѝољубио су имали смисао за мене. Осѝаниѝе увијек заједно, будиѝе ѝоносне на ѝо шѝо јесѝе, и нека Вас кроз живоѝ воде Ваша љубав, доброѝа и разум.

Таѝа вас бескрајно воли!

TABLE OF CONTENTS

ABSTRACT.....	ii
ACKNOWLEDEMENTS.....	iii
1 Introduction.....	1
1.1 Research Motivation	3
1.2 Overview and Scope of Research.....	8
1.3 Structure of thesis.....	10
2 Background.....	11
2.1 Semantics of FMs.....	11
2.2 Manipulating FMs	13
3 Architecture of the FAMILIAR language environment	18
3.1 Overview	18
3.2 The Eclipse Platform as a cornerstone	19
3.2.1 Eclipse Model Framework (EMF).....	20
3.2.2 Graphical Modeling Framework (GMF)	21
3.2.3 Xtext - A DSL Framework	22
3.3 Reasoning Back-ends	24
3.4 Engineering a new FAMILIAR Tool.....	24
3.4.1 Requirements Analysis	25
3.4.2 Design Considerations	26
3.4.3 Implementation Details.....	28
3.5 Usage Example.....	32
4 Evaluation of the FAMILIAR Tool	38

4.1	Study Methodology	38
4.1.1	Goal, Research Questions, and Context.....	39
4.1.2	Hypothesis Formulation.....	40
4.1.3	Experiment Design.....	40
4.1.4	Experiment Objects and Variables.....	41
4.1.5	Scenarios	44
4.2	Experimental Results.....	44
4.3	Threats to Validity.....	50
4.3.1	External Validity.....	50
4.3.2	Internal Validity	51
4.3.3	Construct Validity.....	52
5	Related Work	53
6	Conclusions and Future Work	54
	REFERENCES	57
	APPENDICES	61
A.	Requirements for FAMILIAR Tool.....	61
	Core Functional Requirements	61
	Advanced Functional Requirements	64
	Non-Functional Requirements	65
B.	Evaluation Scenarios	66

1 Introduction

Product line engineering emerged as a result of the profound shift that led from mass production to mass customization [3] in manufacturing industries worldwide. Companies that long embraced the core engineering principle of systematic reuse, by designing parts that can be efficiently assembled to build a product line of related products, managed to successfully produce greater variety and customization in their products. One of the driving objectives for manufacturers has always been to keep improving productivity. They would accomplish this by increasing their own product's functionality and quality while, at the same time, aiming at reducing development costs and time it takes to reach the market.

The real-world examples of successful product line development are everywhere around us. In the telecommunications industry, for instance, Nokia has been building its mobile phone product line with 25-30 new products annually. In comparison, before they adopted product line engineering techniques, they produced no more than 5 new products annually. When Nokia's production team worked on designing products that will be a part of their mobile phone product line, among many others, they considered the following: varying number of keys, varying display sizes, multiple protocols, need for backward compatibility, configurable features, product behavior, post-release changes, 58 languages, and 130 countries [2].

Building a new phone would not require coming up with a brand new design and manufacturing process anymore since a manufacturer can now simply reuse many existing features from its previously released products. New products are now engineered as related "lines" of the same product line.

A product line can be described as a group of products that shares a number of common features and vary only in certain features. The key considerations driving the development of

product lines is how to identify and then reuse this commonality while managing variations in order to reduce the time, effort, complexity, and therefore the overall costs, of creating and maintaining a product line of similar products.

There are many parallels than can be drawn between the manufacturing and software industries. For example, both deal with the same commonality and variability problem, at least on a conceptual level. Similarly, both continuously strive to deliver high quality artifacts using processes with shared attributes such as low production costs and short product development cycle. It is often the case, however, that those activities are governed by conflicting goals.

A notable and ongoing challenge in software engineering discipline is the inherent complexity that begins with a problem domain and carries over to a software system that computes a solution for this problem. Brooks [8] notes that engineers typically deal with two types of complexity: Essential and Accidental complexity. The former complexity is inherent to the problem being solved and cannot be removed. On the other hand, the latter form of complexity is “accidental” to the problem and is more related to the choice of our approach and actions when working on a solution.

Models can help us to break down a complex problem through abstraction. Furthermore, we can also use them to get closer to being able to; ideally, automatically generate a program code from its model, or at least to narrow the gap between the problem and solution domains. Attempts to bridge the problem-solution gap with traditional software development approach are not only labor-intensive but also are tedious and error-prone processes that raise accidental complexities. As a consequence, software developments costs as well as time-to-deploy tend to keep increasing [6]. One of the central ideas behind Software Product Line Engineering (SPLE) is to try to shift away from designing software products separately from all the code that follows.

Instead, our engineering mind should focus on creating quality models that would, with the help of evolving automation technologies and an emergence of widely supported industry standards, eventually be capable of delivering high quality final product in less time and with no or minimal accidental complexities [9].

SPLE has emerged as a promising way to improve the software design and development process by introducing the key aspect of product line discipline which is based on the explicit modeling of what is common and what differs among software product variants. Under the umbrella of SPLE, several perspective approaches have been proposed including code generation [41], components transition [42] and model transformations [43], to name a few.

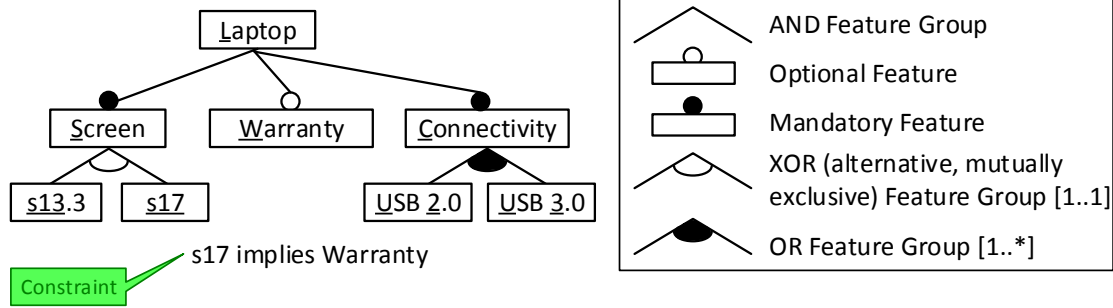
The SPL process involves a significant shift in software production. When decomposing a system in terms of the features it provides, one of the main objectives of SPLE is to construct a well-structured product line which is typically represented with a Feature Model (FM).

1.1 Research Motivation

Feature Models are widely used. Feature modeling is a popular model-driven approach which gives a means to define commonalities and variabilities of a family of (software) products in terms of features. A feature is any distinctive user-visible aspect, or characteristic of a system [18]. For example, it can be functionality of a software system that satisfies a requirement or it can represent a potential configuration option.

The feature model depicted in Figure 1 represents a simple laptop family. An FM hierarchically structures features and feature groups, in a tree-like top-down fashion, using parent-child relations. A feature diagram is simply a graphical representation of an FM typically represented as an And-Or tree with nodes as features. An FM can include constraints that further clarify dependencies among features.

Feature Diagram



Textual Notation

```
Laptop : Screen [Warranty] Connectivity; // Warranty is optional
Connectivity : (USB 2.0 |USB 3.0)+; // Or Group
Screen : (s13.3|s17); // Xor group
s17 -> Warranty; // Constraints
```

For brevity, feature names used below are abbreviated to combination of letters, underlined in the Feature Diagram given above.

A valid configuration of an FM vs. SPL

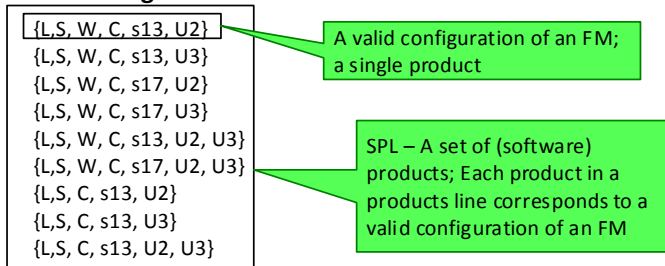


Figure 1: Example of a Laptop FM with different FM notations.

From the example given above, an FM can be represented in several ways: as a feature diagram, or a textual form. Transforming a model from one representation to another always preserves the hierarchy and constraints in their original form.

Each product of a SPL corresponds to a valid configuration of an FM. A configuration is obtained by selecting and unselecting features in an FM. A feature model thus defines a set of valid feature configurations. The validity of a configuration is determined by the semantics of a feature model. For example, in Figure 1, screens with sizes 13.3” and 17” are mutually exclusive and cannot be selected at the same time. Similarly, more expensive laptops (i.e., those with 17”

screens) must include a warranty meaning that all of the laptops with 17” screen will also come with the warranty.

The fundamental idea of SPLE is to decompose a software system in terms of the features it provides. The goal of decomposition is to construct a well-structured software system that can be tailored to the needs of different users and the application scenarios. Typically, from a set of features, many different software product lines can be generated that share common features and differ in other features. Features can also be used to analyze, design, implement, customize, debug, or evolve a software system. For example, if we define Microsoft Office 2013 applications as individual features, then a set of top-level features would include Word 2013, Excel 2013, PowerPoint 2013, Access 2013, and so on. Depending on the selected set of features we might be able to generate different software products. For instance, a product named “Office 2013 Home and Student” would include Word, Excel and PowerPoint, but not Access. On the other hand, a product named “Office 2013 Professional” would contain all four applications with additional sub-packages.

As feature models are rapidly emerging as a viable and important systems development tools, they are also becoming increasingly complex. Managing feature models of industrial size is a tedious and error prone process. To manage the complexity of real-world product lines development, there is a need to create a language that is capable of not only creating, updating, and managing FMs, but also separating, relating and composing them while supporting automated reasoning. To meet this requirement of handling large and complex FMs in a scalable way, the domain-specific modeling language (DSML) FAMILIAR was developed [1, 15, 21].

FAMILIAR is an executable scripting language that has the built-in capability to compose and decompose feature models, and also to manipulate and reason about FMs.

FAMILIAR allows stakeholders to describe domain concepts in terms of commonalities and differences within a family of software or product systems. Feature models are typically passed in to the FAMILIAR interpreter in a textual notation. However, the FAMILIAR framework translates this representation to a propositional formula; so that it can verify the validity of a model, its semantics and perform various computations on an FM such as reasoning and composing operations. In addition, FAMILIAR can interpret a script in order to perform a sequence of operations on feature models. Such scripts are reusable [5].

However, the current text-based version of FAMILIAR has several drawbacks, and this thesis focuses on one in particular: Lack of FM visualization. A number of textual feature modeling languages [11, 36, 37, 38], including FAMILIAR [1, 4] have been proposed during the last decade as a practical solution to the SPL modeling challenge. There are many reasons for choosing textual syntax over graphical since it has many advantages on its own. For example, developing a text-based modeling language requires less effort. This approach is particularly appealing when creating a DSML prototype in an academic environment. Typically, a lightweight textual DSML does not require a rich and dedicated modeling tool since there are already established tools that are available for text-based editing, manipulation, formal reasoning, and versioning. In addition, it is easier to achieve better interoperability of textual models among various languages. However, the problem is that such modeling languages usually do not fit well to the context of established SPL tools such as FeatureIDE. A disadvantage of languages that exclusively support text-based modeling arises from the scale of real-world models. Large models can easily grow exponentially to incorporate thousands of features (i.e., the feature model of a Linux kernel [28]). For example, creating a large text-based FM with thousands of features, and inspecting it while looking for inconsistencies or possible

enhancements or simply comparing it to other FMs, might require a significant mental obstacle for a modeling practitioner.

Cognitive research is a scientific discipline that attempts to gain insights on how the human mind analyzes information, creates knowledge, and solves problems. In the context of SPL and feature modeling, information and knowledge are primarily represented in feature models. There is growing evidence of the cognitive power of visualization [26]. A tool support of visualization models can help modeling practitioners amplify their cognition [27]. The increase in cognitive effectiveness leads directly to improved speed, ease and accuracy with which a model representation can be processed by humans [31]. An incentive for using visual notations is the widely-held belief that they convey information more effectively than text, especially to novices [32].

Modeling practitioners would obviously benefit from additional insights when provided with feature model visualizations in intuitive notations they are already familiar with (e.g., a FODA-like notation) [16, 23]. Allowing users to model feature models in its native, tree-like top-down hierarchical notation, should result in improved efficiency, effectiveness, learnability and model quality.

In summary, using the current text-based version of FAMILIAR might result in a limited modeling experience with inadequate overall usability and productivity. Lower user productivity might also be correlated to a higher number of errors or inconsistencies resulting in lower-quality FMs in general.

The goal of this thesis is to not only enhance the FAMILIAR language itself but also to benefit its modeling practitioners by increasing their productivity as well as quality of FMs they work with.

1.2 Overview and Scope of Research

In this thesis, we are motivated by the following question:

- How we can enhance the FAMILIAR language to help modeling practitioners improve quality of their modeling work, especially in terms of the usability context and FM quality?

One technique that can help us tackle this challenge is visualization. Clearly, there is a need for an UI tool that practitioners can use to manage feature modeling with more user efficiency and effectiveness.

As a part of this thesis, we enhanced the FAMILIAR framework, by adding visualization features, without giving up its powerful textual capabilities. The dual coding theory postulates that visual information (e.g., graph-based models) and verbal information (e.g., text-based models) are stored and processed differently via separate mental channels that do not compete with each other [29]. In other words, using text and graphics together to convey information is more effective than using either on their own.

Supporting both textual and graphical notations in FAMILIAR enables users to better build and manipulate large and complex feature models. As users create or analyze a model they often want to visualize the current state of the model through some graphical notation as an alternative to editing the text-based model. While, on the one hand, text syntax is a perfectly valid way to view, edit, and formally reason about feature models, offering many desirable characteristics such as expressiveness, scripting, reusability, compactness, reproducibility and readability, on the other hand, the graphical notation, might bring improved usability and widespread adoption. Ultimately, the new FAMILIAR Tool must handle the text-based model and the visual model simultaneously by keeping them both fully synchronized in real-time.

In addition, we added a configuration editor and a persistence mechanism as a part of the new tool so that work can be preserved and resumed in subsequent modeling sessions.

In summary, we enhanced the FAMILIAR framework by designing and implementing a new, fully integrated, authoring tool with features such as an intuitive UI, a configuration editor, FM persistence and a graphical representation of FMs with the cross-platform support that SPL practitioners can use to better manage feature modeling. To work with the new standalone FAMILIAR Tool, no external tools such as Eclipse, console, or text editors are needed. The new tool may also make FAMILIAR and SPL in general, more accessible to non-experts.

In addition to the application-based component of this thesis work, there is an experimental component. We performed a small scientific evaluation of the new GUI tool (treatment object) while using the existing text-based tool (control object) in order to get answers to our two research questions:

- (RQ 1) Does visualization of feature models help improve the quality of feature models and reduce user-based errors?
- (RQ 2) Does visualization of feature models help modelers to manage/analyze feature models with improved user efficiency and effectiveness?

Finally, it is important to note that the thesis does not address other aspects of SPLE such as generating implementations from FMs, transforming models, developing a compositional DSML framework (i.e., developing the means to separate language concerns in terms of reusable language fragments representing features), or converting FAMILIAR from external to internal DSML.

1.3 Structure of thesis

The remainder of this thesis is organized as follows. Section 2 provides the background needed to understand the work described in this thesis. Section 3 presents the architecture of the FAMILIAR framework and also details the rationale behind the design and implementation of the new graphical SPL research tool. In this section we also explain some of the visualization and interaction features implemented. We conclude Section 3 with a usage example. Section 4 presents a small experiment that evaluates FM quality and usability aspects of the new FAMILIAR Tool. Section 5 discusses related work. Finally, section 6 describes future work and concludes this thesis.

2 Background

In this chapter, we describe the semantics of Feature Models, manipulation operators as implemented by FAMILIAR, and also provide an overview of tools used in this research.

2.1 Semantics of FMs

A feature model structures product aspects, in a top-down fashion, into multiple levels of increasing detail. When de-composing a feature into sub features, the sub features may be optional or mandatory or may form Alternative-, Or-, or And-groups.

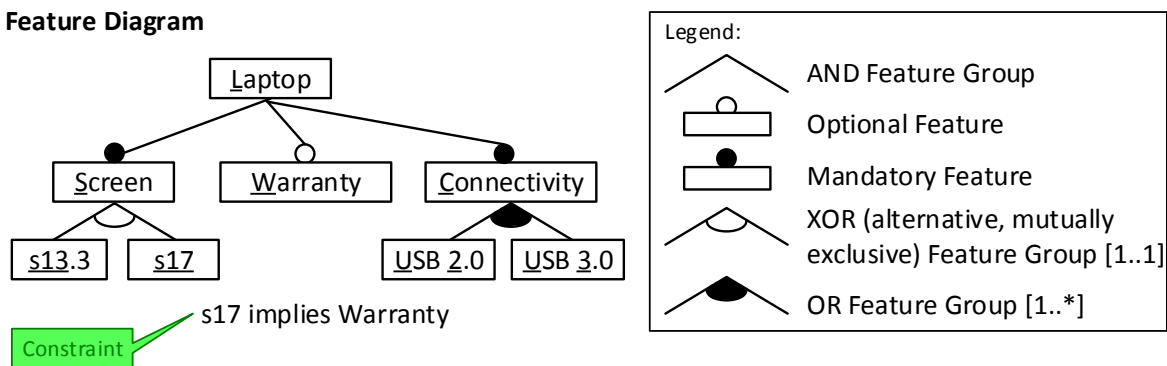
A valid configuration is determined by the following set of rules [1]:

- Any selected feature (node) means that its parent feature (node) is also always selected.
- If a node represents a feature group, and when that node is selected, then the following sub features (child nodes) must also be selected:
 - If a node is an And-group (which may contain mandatory and/or optional features), then **all** of its mandatory sub features are also selected.
 - If a node is an Alternative-group (exclusive OR), then **exactly one** sub feature is also selected.
 - If a node is an Or-group, then **at least one** sub feature is also selected.
- Constraints relating features must always hold.

FM semantics allows one to rigorously reason about feature models by applying Boolean logic. The Boolean expressions simply consist of the constants true (1) and false (0), the operators of conjunction (\wedge), disjunction (\vee), negation (\neg), implication (\Rightarrow) and bi-implication (\Leftrightarrow) as well as propositional variables. A feature model can be converted into a propositional formula. Feature models are translated to a propositional formula through semantic operations

where (1) each feature of the feature model corresponds to a variable of the propositional formula, (2) each relationship of the model is mapped into one or more formulas depending on the type of relationship groups, (3) the resulting formula is the conjunction of all the resulting formulas specified in (2) as well as additional propositional constraints (if any) of the feature model. In his work, Batory [11] explores the use of SAT solvers to reason about feature models whereas Storm [19] considers BDD packages for the same purpose. On the other hand, Czarnecki [10] proposes an algorithm for automatically translating the propositional formula back to a feature model through feature model synthesis (or render operation).

Figure 1 showed graphical and textual representation of an FM. Figure 2a provides an example of the propositional formula for the given feature model of a laptop.



Propositional Formula

$$L_{FM} \leftrightarrow (S \Rightarrow L) \wedge (W \Rightarrow L) \wedge (C \Rightarrow L) \wedge (s13 \Rightarrow S) \wedge (s17 \Rightarrow S) \wedge (U2 \Rightarrow C) \wedge (U3 \Rightarrow C) \wedge // \text{Child-parent}$$

$$(L \Rightarrow S) \wedge (L \Rightarrow C) \wedge // \text{Mandatory}$$

$$(C \Rightarrow U2 \vee U3) \wedge // \text{Or-Group}$$

$$(S \Rightarrow s13 \oplus s17) \wedge // \text{Xor-Group}$$

$$(s17 \Rightarrow W) // \text{Constraints}$$

For brevity, feature names used below are abbreviated to combination of letters, underlined in the Feature Diagram given above.

Figure 2a: Example of a Laptop FM with a propositional formula.

The propositional formula of a feature model is satisfiable if and only if its variables can be assigned to values for which the formula evaluates to true. Otherwise, the propositional formula is not satisfiable. If the propositional formula of a feature model is satisfiable, then all features assigned to True, for any given evaluation, represent a valid configuration. Figure 2b shows an example of a Laptop FM with a valid configuration.

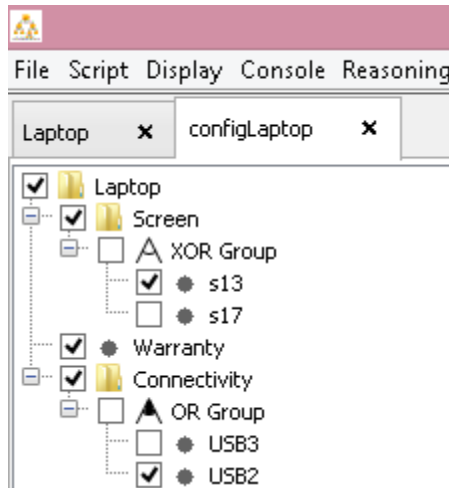


Figure 2b: Example of a Laptop FM with a satisfiable propositional formula. All selected features (set to True in a propositional formula) represent one valid configuration of a Laptop FM.

2.2 Manipulating FMs

One of the most powerful characteristics of the FAMILIAR language is its composition operators that are designed for supporting the separation of concerns in feature modeling. This section provides a brief overview of mechanisms that FAMILIAR uses for composition (i.e., insert, merge, aggregate) as well as decomposition (i.e., slice).

Manually creating FMs is tedious and error prone process. One of the goals of SPL languages and tools is to automate manipulation processes. The resulting FM is formally

synthesized from possibly multiple FMs using underlying semantics. In their work, Thüm et al. [33] identified four possible FM adaptations: (1) refactoring - no new configurations are added and no existing configurations are removed; (2) specialization - some existing configurations are removed and no new configurations are added; (3) generalization - new configurations are added and no existing configurations removed; and (4) arbitrary edits - a change that is none of the above. FAMILIAR supports all four categories of FM adaptations.

Kang et al. [23] introduced the concept of “composition rules” in which “features are related to one another primarily through the use of composition rules, which are a type of constraint on the use of a feature”. It is important to note that although the SPL discipline has offered a multitude of composition approaches, where each one focuses on manipulating artifacts of different types (i.e., code, models, aspects, documents, data types, etc.), FAMILIAR, in its current state, focuses exclusively on the composition of feature models. In FAMILIAR, the semantics of an FM is the set of all valid configurations that contain sets of selected features that respect the dependencies entailed by the diagram and the cross-tree constraints. FAMILIAR thus defines the semantic properties of each (de)composing operator in terms of the relationship among the configuration sets of the input models and the resulting feature model [1].

For composition of FMs, FAMILIAR supports insert, merge, and aggregate operators. The **insert** operator creates a new FM by inserting an input (a.k.a. aspect) FM into another base (a.k.a. target) FM. Other than those two input arguments, the insert operator takes a 3rd argument, an operator mode (i.e., Or, Xor, Opt or Mand) that determines the form of the insertion, that is, whether the insertion preserves the set of configurations defined by an input FM or not. This preservation is the generalization property. The precondition of the insert operation requires that the intersection between the set of features of the base feature model and the one of the aspect

feature model is empty. In other words, it preserves the well-formed property of the composed feature model which states that each feature name is unique. If this precondition is not respected, insert returns false and the base (target) feature model is not modified. Figure 2c shows an example of the insert operation.

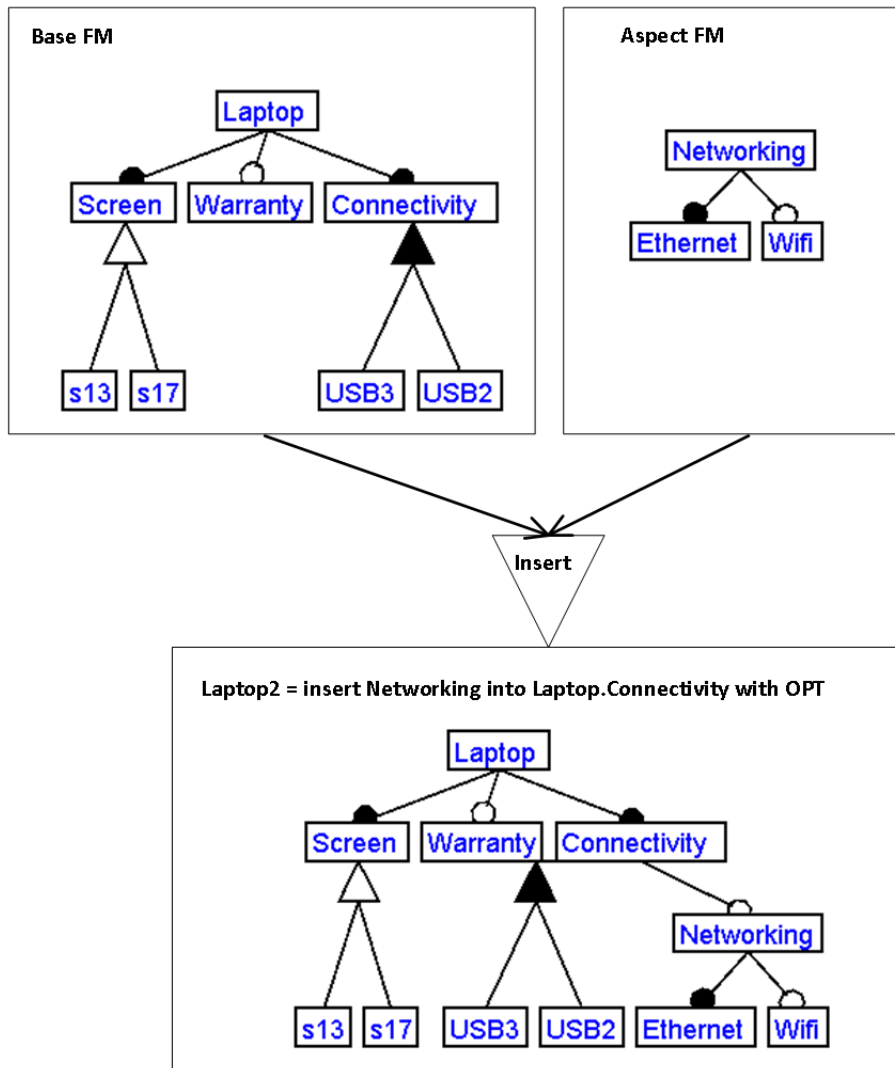


Figure 2c: Example of the FM insert operation with OPT mode.

The **merge** operator is used to combine two or more FMs, and produce a new, integrated, FM. The merge uses name-based matching: two features match if and only if they have the same

name. Several modes are defined for this operator. They indicate how the merge is done in terms of set of configurations preserved in the resulting FM. Similar to the semantics of the insertion operation, the semantics of the merge operator is based on a relationship that exists between the resulting FM and two input FMs. The user is expected to specify this semantics as the 3rd argument: A merge mode, which is either based on the on the union or the intersection of the two input configuration sets. Figure 2d shows an example of two FMs that were created with the merge operator, one with Diff and another one with Union merge mode. In this example, two input FMs, base and aspect, happen to share a feature with the same name, “Connectivity”, but different sub-structures.

Another composing operator that is supported by FAMILIAR is the **aggregate** operator. It is used to inter-relate a set of FMs, eventually with cross-tree, propositional constraints. Contrary to the merge operator, the aggregate operator does not expect a common feature between two input FMs.

When it comes to decomposing FMs, FAMILIAR uses the **slice** operator, which basically produces a feature model that contains only a relevant subset of features.

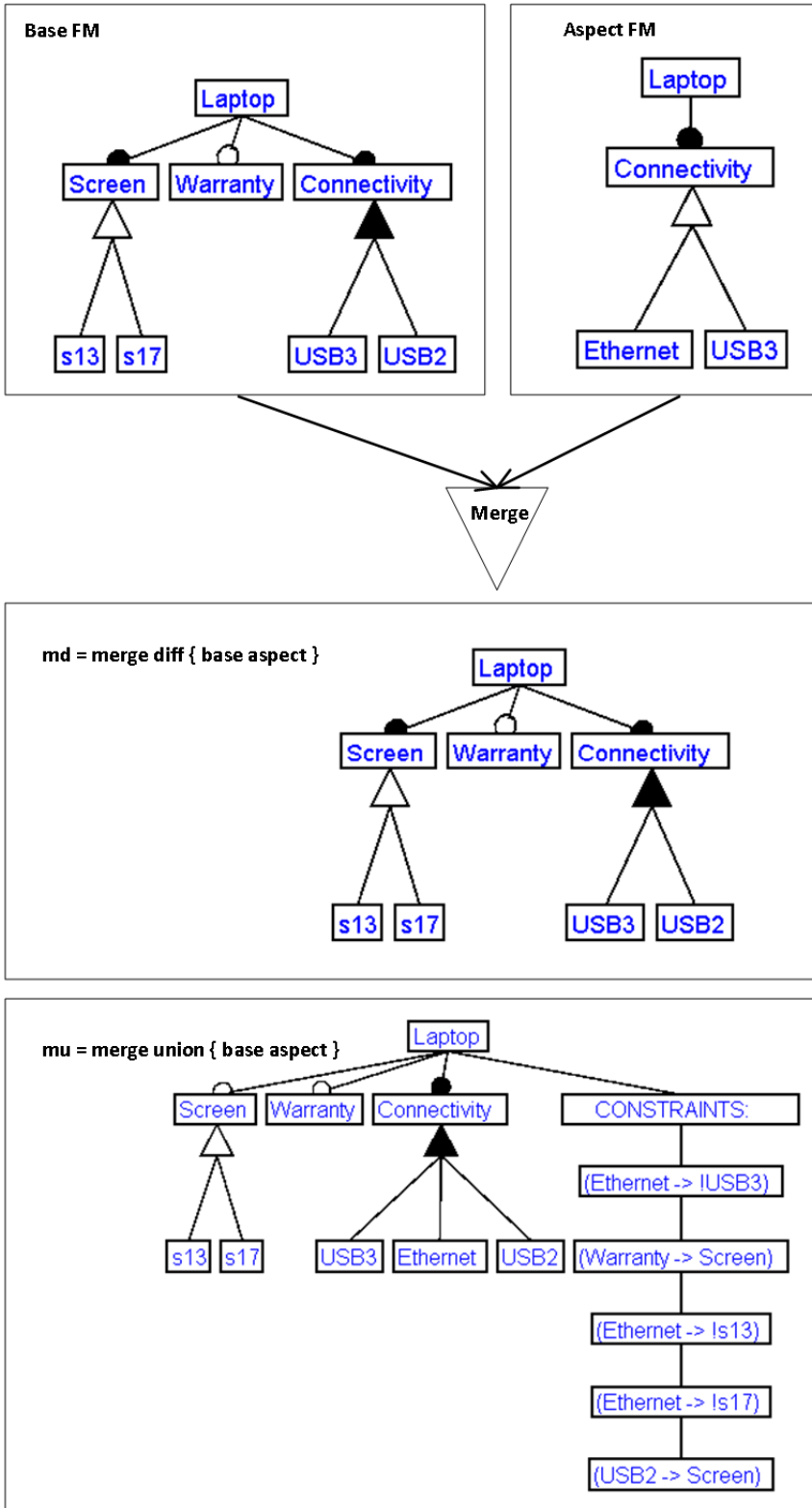


Figure 2d: Example of the FM merge operation with Diff and Union modes.

3 Architecture of the FAMILIAR language environment

In this chapter, we describe the main components of the FAMILIAR language environment - its framework, interpreter, solvers and the existing standalone text-based tool. We then describe the design and implementation of the new FAMILIAR Tool [40, 21] that was developed as a part of this thesis. We conclude the chapter by providing an example of how the new FAMILIAR Tool can be used.

3.1 Overview

FAMILIAR adopts a layered architecture, allowing for extensible design and easier integration with other DSML languages and 3rd party libraries that it uses internally.

As depicted in Figure 3a, FAMILIAR has three main layers

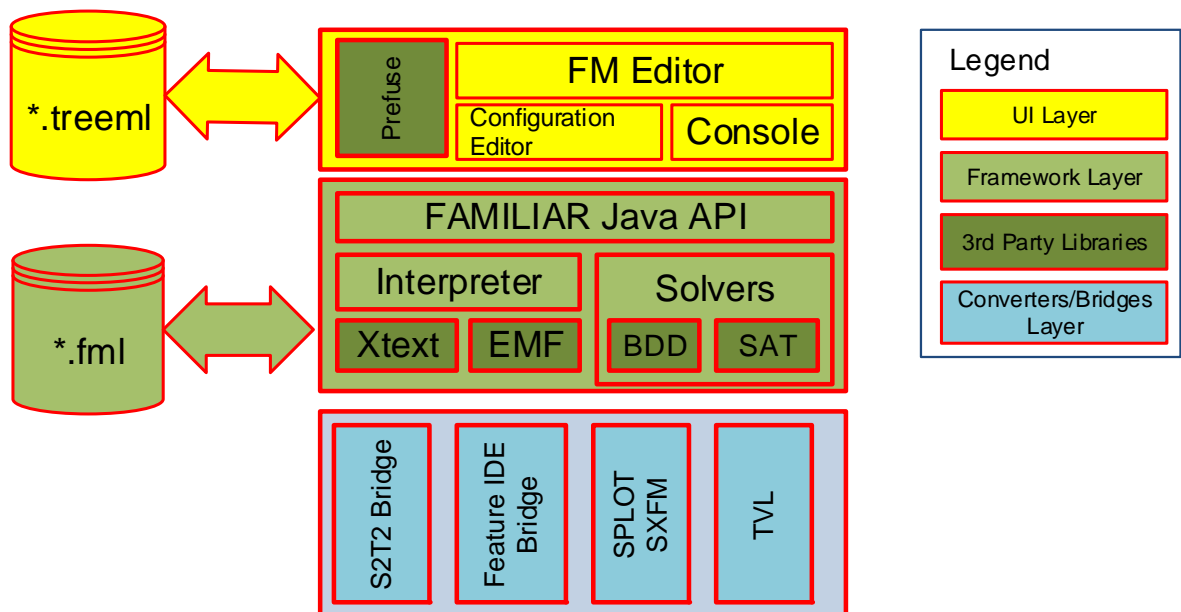


Figure 3a: Architecture of the FAMILIAR language.

- Framework: This is the cornerstone of the FAMILIAR language. The framework specifies the language grammar which allows FAMILIAR to interpret FMs by

building an internal abstract syntax tree (AST) structure. The interpreter uses 3rd party off the shelf solvers (BDD and SAT) to check for satisfiability property of a propositional formula of a feature model. The framework integrates several converters and bridges that allow for integration with other DSML languages and tools. Finally, it exposes its functionality through non-public Java API interface, which is used by both its tools, the visual Editor as well as the text-based console.

- UI Layer: This layer integrates with the Prefuse visualization framework, and exposes the full power of the framework to the end user through FM Editor. FM Editor completely integrates both the Configuration Editor and the Console into unique modeling environment known as FAMILIAR Tool. Console is also available as standalone text-based tool.
- Converters/Bridges: This layer supports several other SPL FM tools and notations.

FAMILIAR's components, shown in yellow in Figure 3a, were developed as part of this thesis work. FAMILIAR Tool leverages all three layers, and provides complete functionality of the FAMILIAR language in an integrated environment.

The next subsections will provide more details about every major FAMILIAR component.

3.2 The Eclipse Platform as a cornerstone

FAMILIAR supports tailoring to specific domains by means of feature model configuration, reusable components, and the reasoning back-ends. It is an executable scripting language that supports manipulating and reasoning about feature models. Moreover, FAMILIAR can interpret a textual script in order to perform a sequence of operations on feature models.

Such operations are reproducible and reusable. In addition, you can import, export, compose, decompose, edit, configure, reason about feature models and combine these operations to realize complex variability management tasks.

FAMILIAR is originally based on Xtext, a popular framework used for creating new DSLs, and then further evolved on the Eclipse platform in Java. FAMILIAR internally uses two off-the-shelf reasoning back-end libraries: SAT4J (SAT solver) and JavaBDD (Binary Decision Diagrams) to support its Boolean-based calculations. Before we developed the graphical editor, FAMILIAR was used either as a standalone (console) application in an interactive mode or as an Eclipse plugin, text-based editor combined with an interpreter that could execute its scripts. In order to boost academic experimentation, the FAMILIAR language supported several notations for specifying feature models including SPLOT/SXFM, FeatureIDE, S2T2, as well as a subset of TVL.

The Eclipse IDE provides a fully integrated and extensible environment well suited for the modeling and building DSMLs such as FAMILIAR. Eclipse integrates Java programming language, Xtext framework, and EMF together with the version control system Subversive (SVN). Eclipse uses plug-in architecture to provide all functionality within the Java runtime system. In fact, the Eclipse IDE products are excellent examples of a software product line with 12 products and 27 features [20].

3.2.1 Eclipse Model Framework (EMF)

The Eclipse Modeling Framework (EMF) is a model management framework implemented atop the Eclipse platform. Eclipse's EMF provides powerful support for defining models and building modeling tools. The core EMF construct is a meta-model (Ecore) component for specifying models and runtime support for the models, including change

notification, persistence support with default XMI serialization, and a reflective API for manipulating EMF objects generically. EMF separates a meta-model from an actual model. A meta-model describes the structure of models. A model is then the instance of a meta-model. Once the meta-model is specified, EMF generates a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor [14]. Ecore models are by default specified in XMI, but they can also be defined using either annotated Java, UML, or XML documents. Both Eclipse and EMF are considered de facto standard technologies in the Model-Driven Development (MDD) community. Using EMF to define the FAMILIAR's meta-model (or domain model) has several advantages. Firstly, it aims to increase productivity and consistency that result from automatic code generation. Secondly, it generates Java classes with clean, simple, and defect-free code. Finally, it supports built-in object persistence and notifications based on the Observer pattern.

The use of EMF also allows us to leverage Graphical Modeling Framework (GMF) in order to build the FAMILIAR editor by using its Ecore model.

3.2.2 Graphical Modeling Framework (GMF)

In addition to EMF, the Eclipse platform offers the GMF. Its main purpose is to enable end-users to generate capable graphical editors for constructing and editing models as defined by the Ecore meta-model. GMF adopts a generative approach to achieving its objective. Its workflow starts with the Ecore meta model which specifies the abstract syntax of the modeling language, and then proceeds with transformations by deriving and maintaining a set of more fine grained, lower-level models that describe graphical syntax and implementation options, and which then can be consumed by the GMF code generator to realize the editor. EMF and GMF are

designed to be used together. They can provide particularly powerful functionality, offering rich customization options for almost every aspect of the generated editor.

3.2.3 Xtext - A DSL Framework

A new DSML language is typically developed with a software tool, or a DSL designed for a development of such languages, or by following a traditional development approach and using a general-purpose programming language of choice. FAMILIAR, which is a DSML, was developed with Xtext, which is a DSL. Xtext [12] is a powerful framework used for the development of external DSLs. Xtext is capable of generating not only a parser but also a semantic model built on EMF, for the Abstract Syntax Tree (AST). It is important to note that Xtext treats a semantic model and AST as the same concept. However, it is better if a semantic model is distinguished from AST [13] since it enables a clear separation of concerns between parsing a language (i.e., the legal expressions of the FAMILIAR program) and the resulting semantics (i.e., what FAMILIAR scripts or commands do when execute).

FAMILIAR's grammar is specified in Xtext's grammar language (figure 3b). The grammar language is a DSL itself designed for the description of textual languages. From this grammar, Xtext produces two artifacts. First, it derives an Ecore model which is an in-memory object graph. The object-graph is an instance of the EMF meta-model. This artifact is used to describe FAMILIAR's concrete syntax and determines how it is mapped to its semantic model. Second, it generates an ANTLR parser and the Java source code for the object model.

```

FeatureModel : ('FM' | 'featuremodel') LEFT_PAREN (
    (
        (root=ID ';' ) | ((features+=Production ';' )+
            (expr+=CNF ';' )*)
    ) | (file=StringExpr) ) RIGHT_PAREN ;

Production : name=ID ':' features+=Child+ ;

Child      : (Mandatory | Optional | Xorgroup | Orgroup | Mutexgroup ) ;

Mandatory  : name=FT_ID ;
Optional   : LEFT_HOOK name=ID RIGHT_HOOK ;
Xorgroup   : LEFT_PAREN features+=FT_ID (B_OR features+=FT_ID)+ RIGHT_PAREN ;
Orgroup    : LEFT_PAREN features+=FT_ID (B_OR features+=FT_ID)+ RIGHT_PAREN
PLUS ;
Mutexgroup : LEFT_PAREN features+=FT_ID (B_OR features+=FT_ID)+ ')?' ;

CNF : Or_expr ;

Or_expr returns CNFExpression:
    And_expr ({Or_expr.left=current} B_OR right=And_expr)*;

And_expr returns CNFExpression:
    Impl_expr ({And_expr.left=current} B_AND right=Impl_expr)*;

Impl_expr returns CNFExpression:
    Biimpl_expr ({Impl_expr.left=current} B_IMPLY right=Biimpl_expr)*;

Biimpl_expr returns CNFExpression:
    Unary_expr ({Biimpl_expr.left=current} B_BIMPLY right=Unary_expr)*;

Unary_expr returns CNFExpression: Neg_expr | Primary_expr ;

terminal LEFT_PAREN : '(' ;
terminal RIGHT_PAREN : ')' ;

terminal B_NOT :      '!' | '~' ; // 'not' |
terminal B_AND :      '&' | 'and' ; //| '&' ;
terminal B_OR :       '|' | 'or' ;
terminal B_IMPLY :    '->' | 'implies' | 'requires' ;
terminal B_BIMPLY :   '<->' | 'biimplies' ;

```

Figure 3b: Part of the FAMILIAR's grammar.

Before the FAMILIAR interpreter executes a script or a command, it first checks whether its statement conforms to the FAMILIAR syntax specified by its grammar. If there are no syntax errors, then the interpreter's semantics is to simply execute Java code attached to grammar elements. They are represented as terminal nodes in FAMILIAR's grammar.

3.3 Reasoning Back-ends

As it was discussed in section 2.1, the idea of transforming feature models to propositional formulas, and then solving the satisfiability problem has been studied by several authors. A propositional formula is satisfiable if it is possible to find a configuration that makes the propositional formula of a feature model true. FAMILIAR uses this approach to reason about FMs using either BDDs or SAT solvers. However, its composing/decomposing operations are currently limited to using BDDs only [1].

3.4 Engineering a new FAMILIAR Tool

As a part of this thesis, we worked on extending the FAMILIAR framework to support its use as a graphical standalone authoring tool. This work had several phases. First, we analyzed the existing architecture of the FAMILIAR framework. As a result of the analysis, the stakeholders agreed that the most optimal course of action would be to extend it in such way to add a GUI layer, an editor application that will be integrated on top of the FAMILIAR framework through its middleware API. Secondly, we worked on gathering the requirements for a new tool. Thirdly, we considered different design approaches, and actually tried two of them: MDD as well as traditional development approach. Our motivation was to assess the current state-of-the-art of MDD tooling using the actual, real-world, product. Fourthly, we worked on implementing an editor in Java using the Prefuse visualization kit. This was an agile driven effort with multiple iterations and test-driven development. There was also notable testing work that was running simultaneously with design and development. During this stage we fixed dozens of newly found or existing issues, and improved the code base in other ways. Finally, we performed an evaluation of the new FAMILIAR Tool, comparing it to the legacy text-based console, to study the usability impact of our new visual SPL tool, especially on novice practitioners.

This chapter describes the development activities. The next chapter describes the evaluation experiment setup, and its results.

3.4.1 Requirements Analysis

Our main goal was to develop a new, user-friendly and easy-to-use, standalone tool which supports (1) visualization of feature models, (2) provides a configuration editor and (3) enables the persistence of FMs. The new tool would provide an integrated modeling environment within the FAMILIAR framework without requiring use of any other IDE (i.e., Eclipse) or plugins. In addition, the tool would still expose all of the original expressiveness, de/composition, reasoning, editing, scripting, interoperability and other facilities of the FAMILIAR language. The complete list of all formal requirements can be reviewed in Appendix A. The following paragraph briefly outlines the most important features of the tool:

- **FM visualization:** Feature models are presented in their basic, propositional form (FODA-Like). The tool supports visual operations such as expanding/collapsing, zooming in/out, zooming to fit, and panning of feature models. In addition, the tool supports fast rendering of even relatively large FMs with 1000+ features, groups and/or constraints.
- **Configuration Editor:** The tool supports creating and editing configurations of feature models. The editing operation enables a user to visually select or deselect features of an FM. The tool checks for a validity of the configuration on-the-fly. In other words, features that are currently selected or deselected represent one valid configuration of an FM. A valid configuration of an FM represents a product in a product line.
- **FM persistence:** Feature models can be saved to an XML file, and loaded from the same file format. This file format is proprietary of the FAMILIAR Tool. In addition, feature

models created with the FAMILIAR Tool can be easily interchanged with several other notations with tool's import and export operations.

- Interpreter: The tool embeds the text-based console that can interactively execute FAMILIAR operations and/or scripts. All commands, regardless of their input mode (i.e., visual or textual) are directed to the same FAMILIAR environment. This provides complete consistency and integrity of loaded FMs during modeling sessions.

3.4.2 Design Considerations

This section presents our two approaches to designing and implementing the FAMILIAR Tool.

3.4.2.1 The MDD Approach

The main focus of MDD approach is to create a model, as a first class development artifact of software and then transform it to produce the source code. The basic idea, as it was described before, is to gradually evolve this abstract model into the final product through a process of incremental refinement, without requiring a change in used methodology or development platform. The advantage of this approach should be self-evident since there are no risk-laden semantic gaps to overcome when transferring a design into production.

Before we began working on the tool, we developed the text-to-model framework (Xtext) that mapped a FAMILIAR's FM textual notation to its abstract syntax model. This was defined by EMF's Ecore meta model, and it represented the FAMILIAR's core model. Since FAMILIAR itself was built on the notion of MDD paradigm, it seemed natural to adopt the MDD approach where we would start with the existing FAMILIAR's Ecore model and then use Eclipse's GMF to transform it to a fully functional graphical editor.

This approach appeared like a good starting step for a number of reasons. Firstly, we already had the Ecore model that we could reuse to initiate the GMF workflow. Secondly, we felt this approach might spare us from the hassle of dealing with the implementation details required to build the editor. This approach would not only boost our productivity but also reduce the accidental complexities that would otherwise be introduced with the traditional development approach. Another benefit would be manifested in a reduced total development time. Our initial estimate was that it would take anywhere from 3 to 4 months for one seasoned engineer to design, implement and test the editor with Eclipse/Java development platform. On the other hand, GMF appeared to be capable of speeding up this whole operation by three or fourfold, even for the same person, who was not previously accustomed to the GMF tooling. Thirdly, the advantage of the MDD approach with GMF is that it uses de-facto standard platform for the construction of the models which supports good interoperability environment. Finally, we wanted to assess the current state-of-the-art of MDD tools on the real world project, and to be able to compare the outcomes of adopting the traditional development vs. MDD approach on the same software project.

As it turned out, this approach did not work well for our particular case. After three weeks of numerous trials, we did manage to transform the initial Ecore model to the complete editor at the end. However, its functionality was extremely limited, both with respect to feature set and performance. This attempt revealed several difficulties when solving the problem of transforming models to code. Firstly, we had the problem of complexity. Each subsequent GMF's phase represents a model with a different (mostly reduced) level of abstraction. Since the GMF workflow requires several models to be produced along the way, until the code gets generated, this increases complex interrelationships among the models making the overall

artifact harder to efficiently produce. Secondly, we found that GMF's model transformation process required lots of hand crafted inputs. Combined with lack of documentation and wider community support, our process quickly resulted in a loosely guided and error-prone effort.

3.4.2.2 Second Approach: Traditional software development

Our second approach was simply to use the traditional software development. We followed two core guiding goals when designing the FAMILIAR Tool:

- Goal #1: Model visualization - Improve cognitive effectiveness for users to enable more effective modeling.
- Goal #2: Model mapping - Achieve a non-ambiguous mapping among three main internal model representations. This implies that model elements of the textual (source) model had to be entirely mapped to model elements of the graphical (target) model without losing their semantic meaning. In addition, the graphical model is serialized to the data storage model.

3.4.3 Implementation Details

Most of the design and development work was completed within four months. The following section provides some of the implementation features.

3.4.3.1 Visualization of Feature Models

We realized that the choice of a visualization toolkit should be made early in our design process since this decision imposes not only visualization techniques but also a data structure to work with. Since our development platform was Eclipse with Java legacy code base, we considered only the visualization kits available on Java platform such are Zest, JUNG, Prefuse, Protovis, SWT, and GEF to name a few. Our design choice was to use the Prefuse Visualization

toolkit. It is an interactive graphical open source library designed to support the development of interactive visualizations. The architecture of Prefuse utilizes the Visualization Pipeline, which decomposes design into a piped process of, firstly, representing abstract data, secondly, mapping data into intermediate, visualizable form, and then finally using these visual constructs to provide interactive views. This improves scalability and representational flexibility. In addition, this separation of concerns provides a degree of flexibility unmatched by existing toolkits, supporting multiple views, semantic zooming, data and visual transformations, and fine grained customizations [30].

Figure 3c below shows the main window of FAMILIAR Tool.

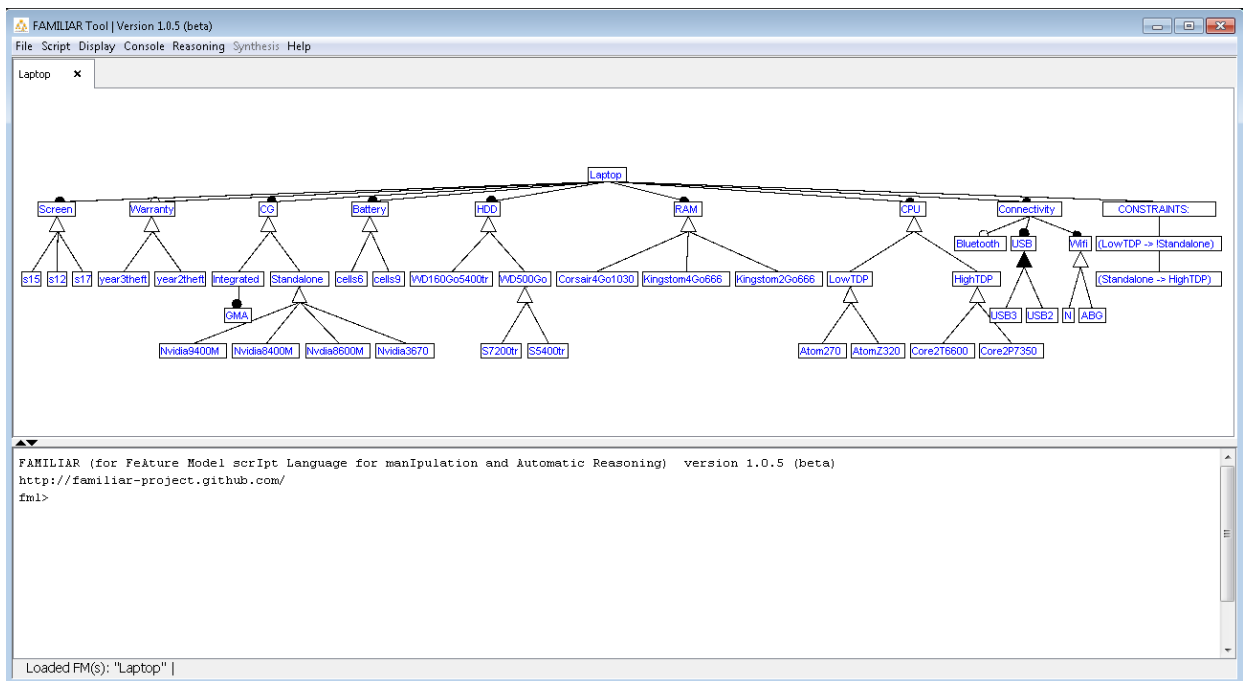


Figure 3c: FAMILIAR Tool - Screen snapshot of main window.

The main window of FAMILIAR Tool has two sections: Visual (upper section) and embedded console (bottom section). FMs that are displayed in visual section are read from and written directly to the FAMILIAR environment. Similarly, textual commands that are issued

either interactively through the embedded console or by running the script, are written to the same FAMILIAR environment. This way, any model update, no matter how it is done (i.e., visually, interactively through command console or through script execution), always keep all of the FMs in a fully synchronized and consistent state. The environment is initialized when the tool is booted, and it is released when the tool is closed.

A user can choose to create a new feature model, or load an existing one. This can be done in several ways. For example, a user can create a new feature model from the scratch. This can be either done interactively with pop-up menu commands, or embedded text commands, or by running a script, or by importing an FM from other SPL tools and/or notations, or by loading a (saved) FM from previous FAMILIAR sessions, or by combining any of above. A feature model is displayed and accessible under a single tab, that is, each FM gets a visualization of its own, and can be modeled independently from other loaded FMs. Executing a script that, among other things, creates several feature models, would create several tabs, each of them containing pre-loaded feature model. Closing a tab would not remove its associated feature model from the environment. For that purpose, a user can run “Console -> Unload FMs”.

We also integrated the existing FAMILIAR’s commands that were used for visualizing feature models through the FeatureIDE plugin (i.e. ‘gdisplay’ command). This was simply achieved through the observer pattern. For example, once the FAMILIAR interpreter detects ‘gdisplay’ statement, it would create an observer handler as well as an observable event source with feature model variable name. Then, it would subscribe the observer handler event to the event source. Finally, this observable event would be handled on GUI level by loading an appropriate feature model variable that corresponds to the given feature model variable name.

3.4.3.2 Persistence of Feature Models

Unlike the text-based tool, the new tool enables feature model persistence. The persistence of feature models is achieved through a serialization of a feature model, from its visual representation to its persistent storage with an XML-like structure. This, in turn, leads to a more complex design since it imposes a constraint that requires maintaining one-to-one mapping among three internal FM model representations: (1) FM environment variable with its associated AST model of a feature diagram, (2) visual FM object with its associated Prefuse interactive view model, and (3) serialized FM to XML storage with its associated in-memory representation.

3.4.3.3 Integration with command line interpreter

Integration with the command line interpreter was simply achieved by forwarding down the system input stream (text-based commands) from the embedded console (GUI control) to the FAMILIAR framework, and by redirecting the system output streams back to the same GUI control. This way, the embedded FAMILIAR console which is part of tool (bottom section of the main window shown in Figure 3c) behaves the same way, syntactically and semantically, as the old standalone text-based tool. In addition, any action that is committed directly through the embedded console control is automatically propagated up to visualized objects.

3.4.3.4 Configuration Editor

The Configuration Editor is implemented as an interactive Java tree control that represents an FM with its set of selected and/or deselected features. A feature is allowed to be selected or deselected only when its FM's propositional formula is satisfiable.

3.4.3.5 Basic Code Metrics

The basic statistics we present here is given for the work committed on the package level only. Any modifications performed on FAMILIAR's API or framework level (i.e., use of the

observer pattern to support ‘gdisplay’ command or simply refactor functionality of the framework to better handle requests from GUI level) is not included here:

- Number of files: 30
- Number of classes: 78
- Number of images: 7
- Lines of code (including comments, no blank lines): ~4,020
- Lines of code (including comments, with blank lines): ~4,560
- Total project size (.java files only): 160KB

3.5 Usage Example

In this section, we demonstrate the core capabilities of FAMILIAR Tool by using a simple example, a Digital Calculator SPL. The example scenario is given below.

Small software company ABC Inc. develops and sells digital calculator software products offering three basic applications: Standard Digital Calculator, Scientific Digital Calculator, and Programmer Digital Calculator. Recently, ABC Inc. learned more about some of the potential benefits of using software product lines so their management decided to enhance its software production process by adopting a SPL engineering practice. After doing research on SPL tools, they choose FAMILIAR Tool to help them with this transition. The FAMILIAR Tool was their top choice since it appeared to have several decent features they were ideally looking for in a visual SPL tool: (1) Built-in FM composition/decomposition operators, (2) SPL configuration editor, (3) FM editing and reasoning capabilities, (4) interoperability with other SPL tools and FODA-like notations, (5) scripting and (6) integrated standalone SPL authoring tool, easy to use and learn.

The modeling task is assigned to a senior software engineer in their company who is familiar with the architecture of Digital Calculator’s software product line. The SPL modeler begins using the FAMILIAR Tool. First thing that he notices after booting the tool is that it preloads an example of a generic Laptop feature model. He appreciates this template since it allows him to get started quickly without prior exposure to the tool itself. He performs several basic editing and reasoning ad-hoc operations trying to become more familiar with tool’s core features (Figure 3c).

The modeler then proceeds by creating three new feature models for their existing Digital Calculator applications (product line). He uses context-sensitive pop-up menus to build up new feature models. While doing this work he is careful to use the same root name for all three feature models so that they can easily be merged and then configured in subsequent steps. By leveraging his knowledge of the architecture of Digital Calculator applications, he first creates an FM of standard calculator. Each node and feature group in his model represents the real subsystem, class or module of their software. The modeler then creates two remaining FMs while being careful to use the same names for all features that share the same code base.

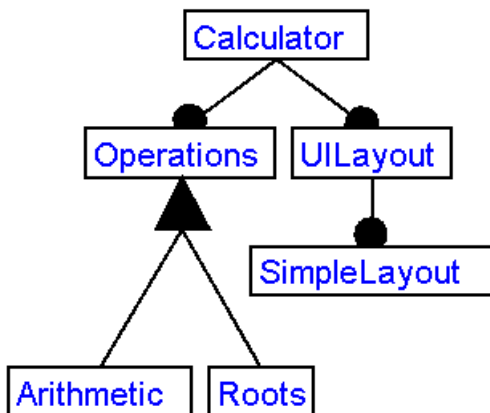


Figure 3d: Feature model of Standard Calculator

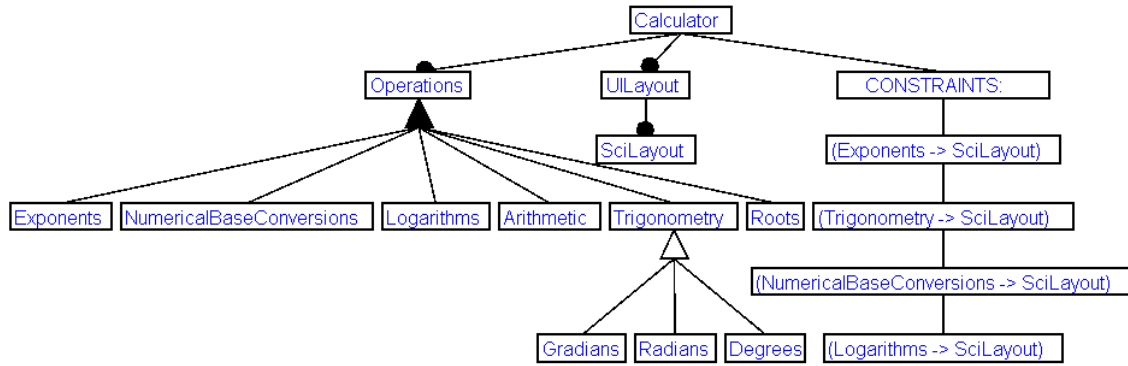


Figure 3e: Feature model of Scientific Calculator

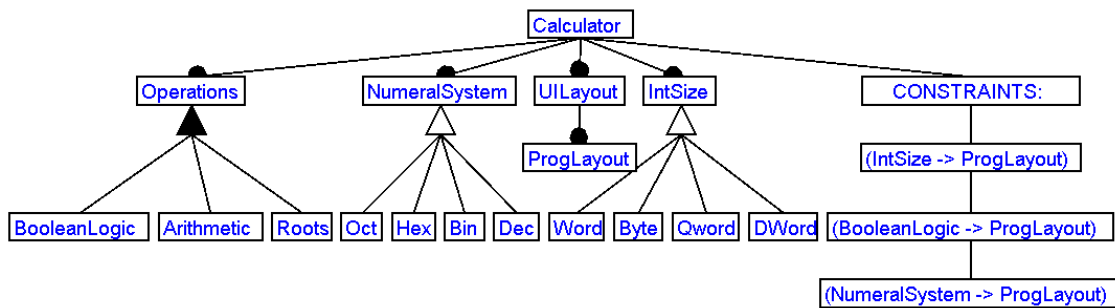


Figure 3f: Feature model of Programmer Calculator

The modeler then decides to save his work so that it can easily be modified and revised in future modeling sessions. He proceeds by saving each feature model to a file “File -> Save FAMILIAR FM As ... (*.treml)”. However, he realizes that he can further improve productivity and consistency of his modeling work by automating the tasks that he had already executed so far. This reproducibility might come handy particularly in cases when more modeling revisions might be required to further fine tune or even revise the existing feature models. Rather than manually reloading every FM individually, the reproducible semi-automated scenario can be easily achieved with a FAMILIAR script that would use text-based syntax of each FM to construct (and load) new FM objects. To create this script, the modeler first needs to get a FM text-based description from its visual notation. For each FM, he navigates to “Reasoning -> Textual Syntax”, and then copies the resulting string to a file that he named “Calc.fml”. While

doing this he also makes sure to assign a unique feature model variable name (i.e. standardCalc) to each generated FM textual definition:

```
standardCalc = FM ( Calculator : Operations  UILayout  ;
    UILayout : SimpleLayout ;
    Operations : (Arithmetic|Roots)+ ; )

sciCalc = FM ( Calculator : Operations  UILayout  ;
    UILayout : SciLayout  ;
    Operations :
    (Exponents|NumericalBaseConversions|Logarithms|Arithmetic|Trigonometry|Roots)
+ ;
    Trigonometry : (Gradians|Radians|Degrees) ;
    (Trigonometry -> SciLayout) ; (Exponents -> SciLayout) ;
    (NumericalBaseConversions -> SciLayout) ; (Logarithms -> SciLayout) ; )

progCalc = FM ( Calculator : Operations  NumeralSystem  IntSize  UILayout  ;
    UILayout : ProgLayout  ;IntSize : (Word|Byte|Qword|DWord)
;NumeralSystem : (Oct|Hex|Bin|Dec) ;
    Operations : (BooleanLogic|Arithmetic|Roots)+ ;
    (IntSize -> ProgLayout) ; (BooleanLogic -> ProgLayout) ; (NumeralSystem
-> ProgLayout) ; )
```

Figure 3g: FAMILIAR script which creates three Calculator feature models

Running this script at any given time, “Script -> Run FAMILIAR Script (*.fml)”, will simply construct, load and then visualize all three feature models in a current FAMILIAR environment.

The next step is to generate a final model, a Digital Calculator SPL, by composing three existing FMs (i.e., using the ‘merge’ operator):

```
calcSPL = merge union { standardCalc sciCalc progCalc }
```

Figure 3h: Using ‘merge’ operator to combine several FM into the single one

The next figure shows the resulting Digital Calculator SPL:

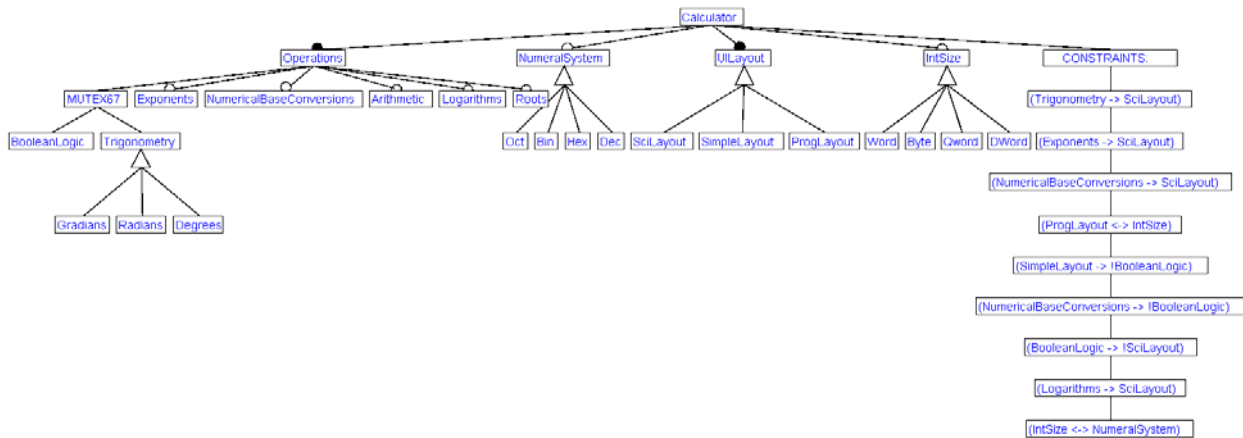


Figure 3i: Digital Calculator SPL of the ABC Inc. company

The modeler then decides to verify validity of this newly composed FM by running some of the formal reasoning operators (i.e., check for valid configurations, dead features, etc.).

Finally, the modeler decides to create a configuration of Digital Calculator SPL. This is simply done by right click on the root node and then selecting ‘New Configuration’. Figure 3j shows the outcome:

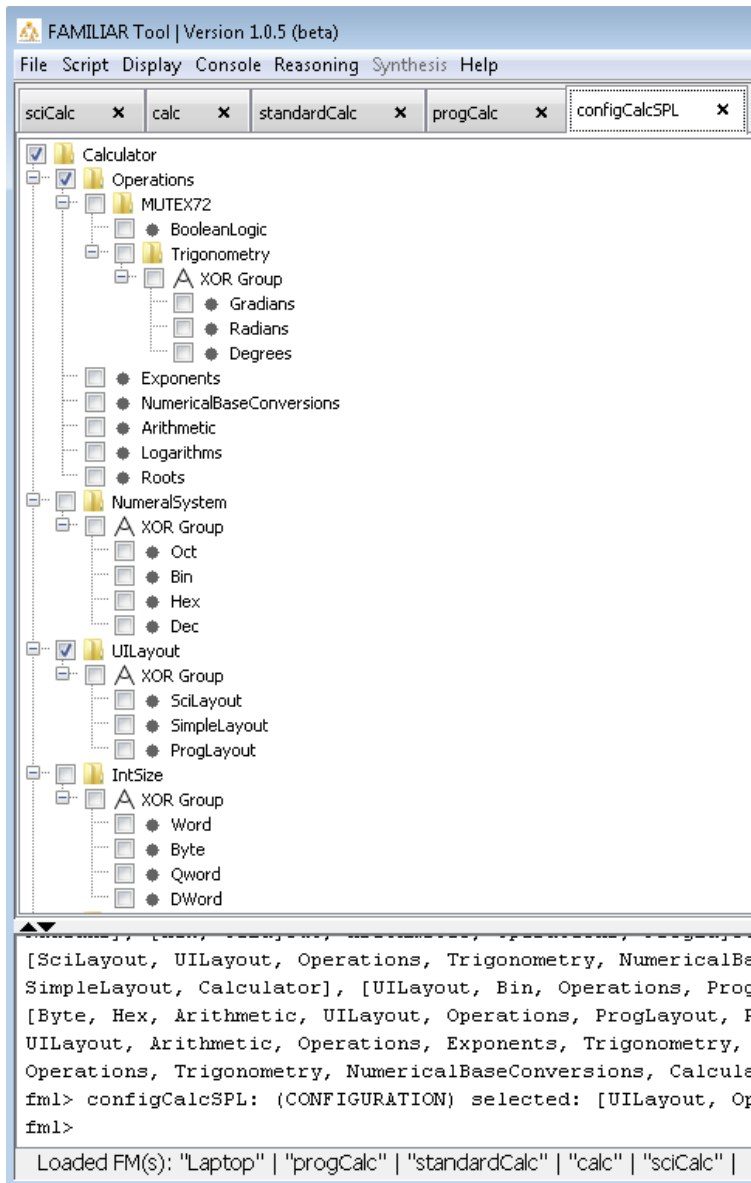


Figure 3j: Configuration of Digital Calculator SPL

The modeler can now start analyzing and configuring the Digital Calculator SPL. For example, he wants to find an answer to a question: “What work would be required to produce a new product called Statistics Digital Calculator”? Or “How we can refactor our existing architecture or design of our product line to speed up delivery of new releases without giving up on quality and feature set”?

4 Evaluation of the FAMILIAR Tool

In this chapter, we describe the evaluation of our new FAMILIAR Tool [40, 21]. In particular, we present specific features of the small scale lab experiment we performed in order to identify differences when using our newly developed, GUI-based, FAMILIAR Tool (with visualized feature models) vs. legacy standalone console (with text-based feature models) by measuring the usability aspects. Furthermore, we outline our investigative approach used to plan, execute, and analyze the evaluation data. Finally, we report on immediate results and interpret them according to the MUSIC methodology [24].

4.1 Study Methodology

We want to investigate the usability of our new modeling tool. The challenge arises from the fact that usability does not exist in any absolute sense. Rather, it would make sense only to define it with reference to particular contexts. ISO 9241 [25] defines usability in terms of the quality of use as the “effectiveness, efficiency and satisfaction with which specified users achieve specified goals in particular environments”. Bevan [24] uses this standard to describe a method called MUSiC (Metrics for Usability Standards in Computing) for specifying the context of use when measuring user effectiveness, efficiency, and satisfaction. The context need to define who the intended users of the system are, the tasks those users will perform with it, and the characteristics of the organizational or social environment in which it will be used. This method seems particularly suitable for our evaluation since (1) it focuses on usability attributes that we are interested in measuring, and (2) it relies on ISO 9241 which appears to be widely adopted standard industry wide.

Evaluating the FAMILIAR Tool would be possible to perform in such a way that provides us with a high level of control over the variables that can affect the study outcome.

Similarly, by running the tool in order to accomplish and measure certain scenarios it would be possible to achieve a high level of replication in an environment where both the difficulty of control and the cost of replication are fairly low. As such, we propose using a formal experiment as our primary type of study.

4.1.1 Goal, Research Questions, and Context

We formulate the goal of the FAMILIAR Tool evaluation using the Goal-Question Metric (GQM) template [17] as follows:

- Evaluate the FAMILIAR Tool to better understand the impact on usability aspects of implementing feature model visualizations on the FAMILIAR language from the viewpoint of modeling practitioners.

Based on this goal, we focus on the following research questions:

- RQ1: Does visualization of feature models help modelers to author FMs of a better quality?
- RQ2: Does visualization of feature models help modelers to manage/analyze feature models with better efficiency?
- RQ3: Does visualization of feature models help modelers to manage/analyze feature models with better effectiveness?

The context selection represented situations where researchers and SPL/MDD practitioners perform feature modeling, in particular, working on creating new FMs. The controlled experiment will be conducted within two groups of graduate Computer Science students with a total of 17 participants from two countries, United States and France. Our first group involves 3 graduate students that are under Dr. France's supervision at the Colorado State University (CSU). Our second group involves 14 Master and PhD students that took a graduate

SPL course taught by Dr. Philippe Collet, at the University of Nice Sophia Antipolis (UNSA), France. We decided that UNSA participants, since it is a larger group, form a treatment group by working only with the new FAMILIAR Tool (Visual). On the other hand, CSU participants will form a control group by working only with the legacy standalone tool (Text-based).

4.1.2 Hypothesis Formulation

Presenting models in a visualized form helps the user grasp the information landscape more quickly and intuitively than presenting models in textual form.

- Hypothesis: Using the FAMILIAR Tool with visualized feature models yields higher FM quality, user effectiveness, and efficiency than using the same tool with text only mode, when creating new FMs.

4.1.3 Experiment Design

Participants will be asked to go through several stages before they run the experiment. These stages will involve the following steps: (1) a basic training on SPL, Feature Modeling, and the FAMILIAR language, (2) a preparation for the experiment, and (3) an experiment session.

Firstly, we will provide minimal overview of SPL, Feature Modeling, and brief introduction to the FAMILIAR language and its environment. The training provided will be at the very basic level, and we expect that students do not spend more than one hour before starting the experiment sessions. Secondly, preparation for the experiment will, among the other things, involve getting the FAMILIAR environment properly configured. Finally, the experiment session should last no longer than 55 minutes, and it will consist of two sub-tasks (3.1) analyzing the online configurator for Audi cars, and then (3.2) modeling it by creating a new FM file for the Audi configurator. Students will be asked to work at their own pace, independently of one another. They will also be required to record all of their interactions with the tools during the

experiment session. By doing this, we can uniformly measure number of modeling tasks that users successfully completed, the accuracy with which users completed tasks (i.e., some quantification of errors), the duration of tasks, users’ learning of the interface, and finally, assess quality of FMs created, and calculate user efficiency and effectiveness.

4.1.4 Experiment Objects and Variables

In our experiment, we have one independent variable and three target variables. Since V group is compared to T group regarding its FM Quality as well as user effectiveness and efficiency, **choice of the used tool** could be called the independent variable and **FM quality**, **user effectiveness**, and **user efficiency** are all the dependent variables. The treatment object is the group that uses new FAMILIAR Tool (GUI) and works with visualized FMs. The control object is the group that uses a legacy FAMILIAR console and works with textual FMs.

Table 4a: Experiment variables.

Independent (State) Variable	Dependent (Response\Target) Variables	Scale of Measurement
<ul style="list-style-type: none"> • Used Tool Group V – Works with visualized FMs. Group T – Works with textual FMs. 	<ul style="list-style-type: none"> • FM Quality 	Ordinal
	<ul style="list-style-type: none"> • User Effectiveness 	Ordinal
	<ul style="list-style-type: none"> • User Efficiency 	Ordinal

4.1.4.1 Dependent Variable - FM Quality

According to the MUSiC method, a quality is a measure of the degree to which the output achieves the task goals. For the purpose of this experiment, the quality is expressed in terms of a final FM quality. Table 4b breaks down the criteria that we are going to use to assess the quality of an FM:

Table 4b: Experiment variables.

FM Quality	Assigned Weight	Description
Poor quality	0.2	FM cannot be properly parsed by the tool (i.e., model contains an inconsistent and/or an invalid element(s), or simply it is not syntactically well-formed).
Satisfactory quality	0.4	FM is properly loaded by the tool but lacks majority of features and/or groups.
Good quality	0.6	FM is mostly complete (i.e., includes various Audi model lines) and has neither inconsistencies nor invalid elements.
Very good quality	0.8	FM includes comprehensive features set but might fail to accurately represent certain group dependencies (i.e., used AND-group when XOR-group would be more appropriate).
Excellent quality	1.0	FM includes comprehensive and diverse features set, and provides solid foundation for further breaking down the model as an SPL artifact. Different feature groups, dependencies and constraints were used in terms of both quantity and quality. (i.e., this FM, if offered with an online configurator, has enough details to allow a customer to pick up a model of a custom Audi car tailored for her needs).

FM quality is expressed as a numerical value between 0.2 and 1 where 0.2 describes an FM of the lowest quality, and 1 stands for an FM of the highest quality. The FM Quality uses the ordinal scale of measurement.

4.1.4.2 Dependent Variable - User Effectiveness

To assess user effectiveness, we'll also need to define a quantity. Quantity is basically a measure of the amount of a task completed by a user. It is defined as the proportion of the task goals represented in the output of the task. For the purpose of this experiment, the quantity reflects a measure of FM completeness in terms of a number of features (#F), number of

constraints (#C), valid configurations (#VC) and FM depth (#D). Each of those four categories contributes 25% of a total quantity value. Table 4c shows how we can calculate quantity as a measure of FM completeness:

Table 4c: Experiment variables.

FM Completeness Category	Assigned Weight	Description
Number of features	0.25	20-29 features
	0.5	30-39 features
	0.75	40-49 features
	1	50 or more
Number of constraints	0.25	1 constraint
	0.5	2-3 constraints
	0.75	3-4 constrains
	1	5 or more
Valid configurations	0.25	Up to 100 configurations or above 25k
	0.5	Up to 4999 configurations
	0.75	Up to 9999 configurations
	1	Between 10k and 25k
FM depth	0.25	Depth of 1
	0.5	Depth of 2
	0.75	Depth of 3
	1	Depth of 4 or more

Quantity is then simply calculated as $(\#F + \#C + \#VC + \#D) / 4$. It is expressed as a number between 1 and 100 where 1 represents the least complete FM, and 100 represents the most complete FM. Note that the quantity measure does not (and it should not) reflect the quality of an FM. Finally, the user effectiveness is given as a percentage number, and is calculated with the following formula:

$$\text{User Effectiveness} = (\text{FM Quantity} \times \text{FM Quality}) \%$$

4.1.4.3 Dependent Variable - User Efficiency

Note that the user effectiveness does not take into account a time required to complete a given task. On the other hand, the user efficiency calculation does include the time component.

The user efficiency is calculated with the following formula:

$$\text{User Efficiency} = \text{User Effectiveness} / \text{Scenario Time}$$

Therefore, the user efficiency measures the user effectiveness in terms of time it takes to complete a task. The higher this number is the user is more efficient relative to other users. Note that both user effectiveness as well as user efficiency also use the ordinal scale of measurement.

4.1.5 Scenarios

The complete list of evaluation tasks that were given to the participants is presented in Appendix B. Since not all students completed all of the steps, we'll be considering only Task #1 which asks participant to create a partial feature model supporting the whole structure of the AUDI configurator.

4.2 Experimental Results

We used the Small Stata 12.1 package to perform a statistical analysis and chart all of the graphs shown in this section. Table 4d shows the main experiment data:

Table 4d: Summarized experiment results.

Group	Time (min)	# of user errors	# of features created	FM quality	User Effectiveness	User Efficiency
V	24.8	10	38	0.8	55%	0.53
V	11.4	1	23	0.4	23%	0.48
V	18.6	3	44	0.4	18%	0.23
V	16.2	3	37	0.4	13%	0.19
V	45.2	42	48	1.0	75%	0.40
V	25.3	12	39	0.4	18%	0.17
V	20.2	8	26	0.4	15%	0.18
V	29.0	12	71	0.6	38%	0.31
V	12.4	4	37	0.4	23%	0.44
V	19.8	4	54	0.6	34%	0.41
V	13.6	0	46	0.6	30%	0.53
V	6.8	7	27	0.4	10%	0.36
V	5.8	3	27	0.4	18%	0.72
T	30.0	1	51	0.2	15%	0.12
T	9.9	5	17	0.4	15%	0.36
T	16.4	25	40	0.2	13%	0.18

The last three columns (dependent variables) are determined by the study methodology as it was described in section 4.1.4.

Let's first visualize the data. Figure 4a shows the box plots of all three dependent variables FM Quality, Effectiveness and Efficiency, grouped by the tool used (T and V). Looking at the box plots for the two groups we can observe that T-group has no outliers, and V-group has 2. The medians are indicated by the red diamonds. Note that the median values for all dependent variables are higher for V-group than T-group. Each of the box plots illustrates a different skewness pattern. It appears that the Effectiveness in particular exhibit the non-symmetric distribution which might imply non-normality data.

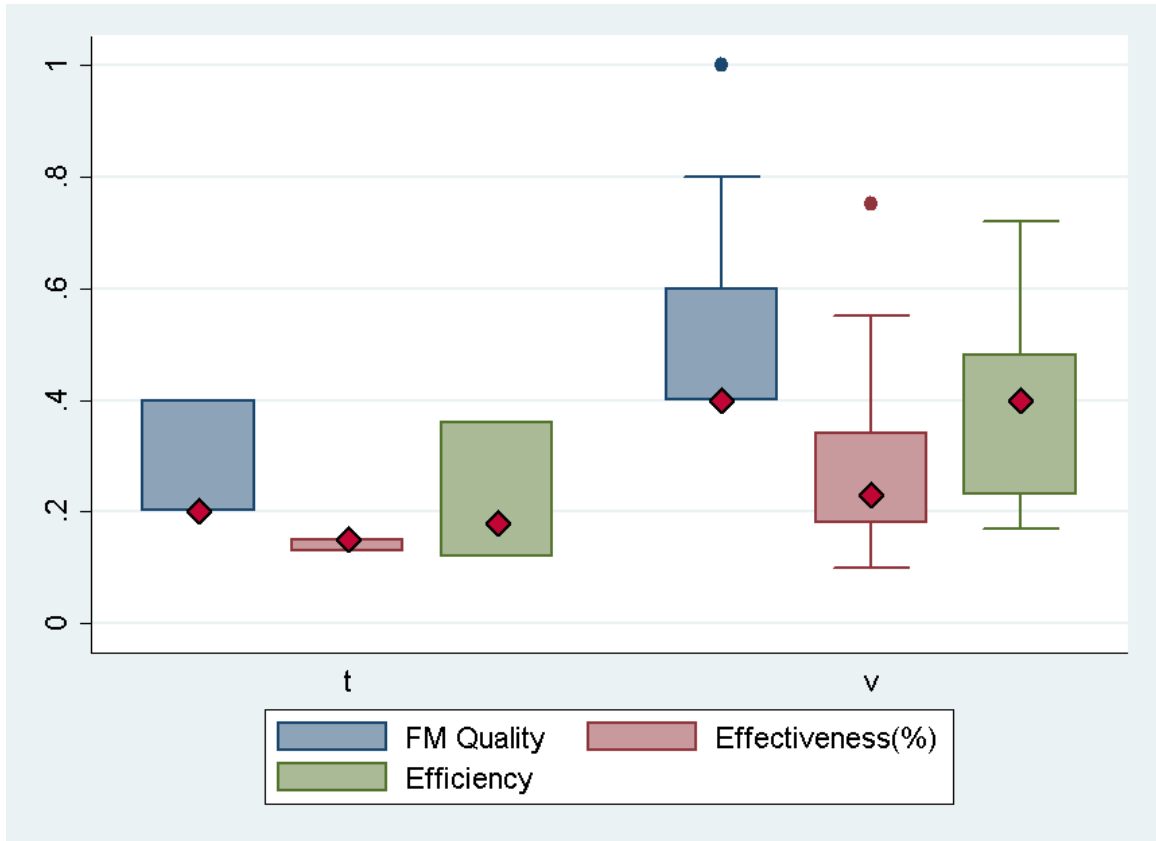


Figure 4a: Box plots of dependent variables, by group

In addition to the graph shown above, Table 4e shows the calculated median values for all dependent variables by group:

Table 4e: Median values of dependent variables, by group.

Group	FM quality	User Effectiveness	User Efficiency
V	0.4	23%	0.40
T	0.2	15%	0.18

We are not concerned with averages since they do not make sense for ordinal data. Based on our small sample data, it is clear that participants who used the visualized SPL tool did better

than participants who used the text-based tool in terms of all three categories: FM quality, user effectiveness and user efficiency. For example, 2 out of 3 participants from T-group failed to produce a valid FM, whereas all of 13 participants from V-group produced valid FMs that could be independently verified after the experiment was completed. However, we need to determine whether this difference in medians between two groups is statistically significant before we can come up with the experiment conclusions.

Figure 4b shows correlation between a number of features in an FM and its impact on an FM quality. It appears that, for the text-based tool, more features in a model tend to reduce an overall quality of an FM. This might make sense since the larger textual models require increased cognitive efforts on the user side. However, the visual tool demonstrates the opposite tendency: More features in an FM in general lead to a slight linear increase of a FM quality.

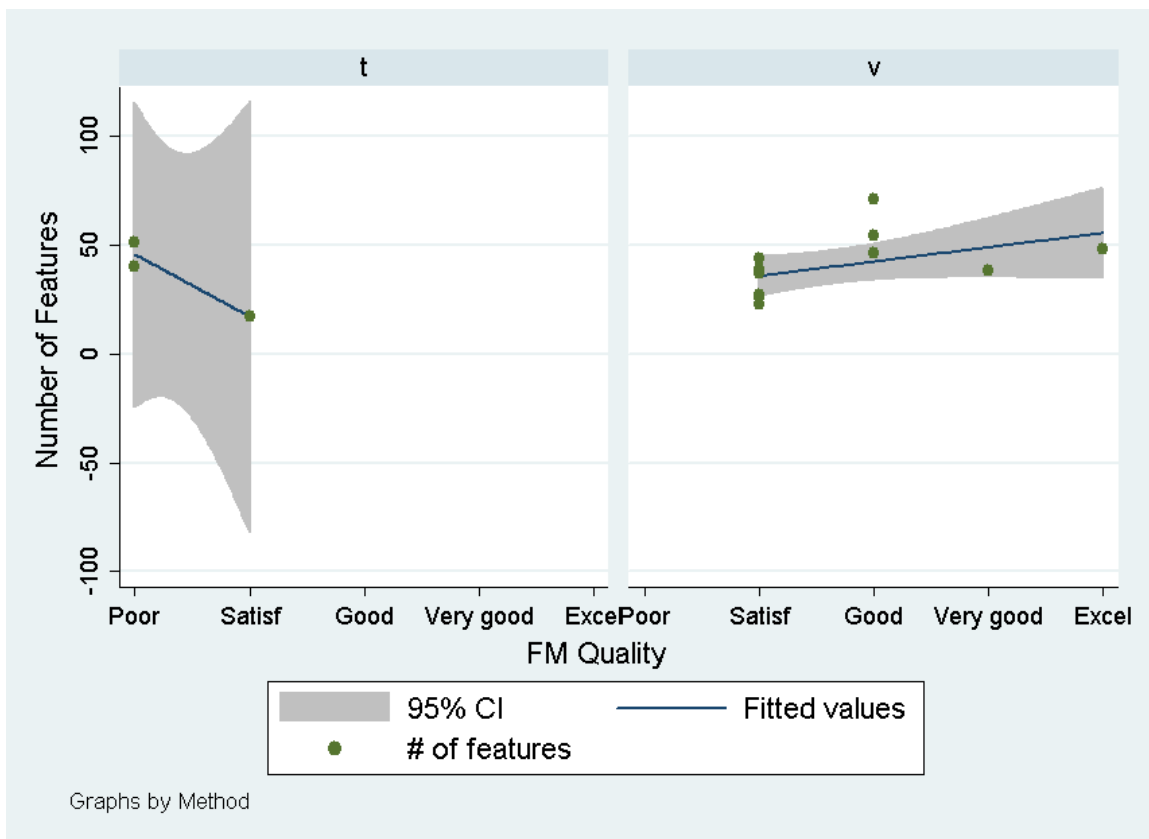


Figure 4b: Scatter plot that correlates a number of features in an FM and its impact on a FM quality

Next question that we need to answer is whether the data come from normal distributions? Unfortunately, our samples are very small. We are going to use the Shapiro-Wilk W test for normal data with a P-value of 0.05 as a cutoff:

Table 4f: Shapiro-Wilk W test for normal data.

variable	obs	W	V	z	Prob > z
fmquality	16	0.93052	1.408	0.679	0.24841
effectiveness	16	0.77929	4.472	2.975	0.00146
efficiency	16	0.95110	0.991	-0.018	0.50731

The lower the P-value is, the smaller the chance that the sample data comes from a normal distribution. Table 4f shows that only the P-value of Effectiveness is lower than 0.05, which means that its sample deviates from normality.

Our experimental design uses one independent variable with two levels (independent groups V and T). In addition, the scale of measurement for all three dependent variables is ordinal with mixed picture when it comes to distribution of the data. Because of all of this, we decided to use the Mann-Whitney Rank Sum Test (also known as Wilcoxon Rank Sum Test) for the statistical analysis of the experimental results. It is a non-parametric test for comparing two groups, and as such, it implies testing analysis that does not assume anything about data normality. Essentially, the Rank Sum Test attempts to provide a statistical answer to a question of whether the two population distributions are different. Another advantage of this test is that is not sensitive to outliers. This is important consideration for the experiment analysis, since it

relies on a very small population sample (13 from V-group and 3 from T-group, 16 participants in total) with 2 outliers.

Tables 4g-4i show the outcome of the Mann-Whitney Rank Sum Tests.

Table 4g: Two-sample rank-sum test for the FM quality variable.

method	obs	rank sum	expected	unadjusted variance	55.25
t	3	10	25.5	adjustment for ties	-10.16
v	13	126	110.5	adjusted variance	45.09
combined	16	136	136		

Ho: fmquality(method=t) = fmquality(method=v)
 z = -2.308
 Prob > z = 0.0210
 P{fmquality(method=t) > fmquality(method=v)} = 0.103

Table 4h: Two-sample rank-sum test for the user effectiveness variable.

method	obs	rank sum	expected	unadjusted variance	55.25
t	3	12.5	25.5	adjustment for ties	- 0.81
v	13	123.5	110.5	adjusted variance	54.44
combined	16	136	136		

Ho: effectiveness(method=t) = effectiveness(method=v)
 z = -1.762
 Prob > z = 0.0781
 P{effectiveness(method=t) > effectiveness(method=v)} = 0.167

Table 4i: Two-sample rank-sum test for the user efficiency variable.

method	obs	rank sum	expected	unadjusted variance	55.25
t	3	13	25.5	adjustment for ties	- 0.24
v	13	123	110.5	adjusted variance	55.01
combined	16	136	136		

Ho: efficiency(method=t) = efficiency(method=v)
 z = -1.685
 Prob > z = 0.0919

$$P\{\text{efficiency}(\text{method}=\text{t}) > \text{efficiency}(\text{method}=\text{v})\} = 0.179$$

The rank sum tests the null hypothesis that two independent samples are from populations with the same distribution. With only 16 observations, the departure would have to be substantial to reject the uniform null hypothesis. We used the "porder" option of the rank sum command to calculate this departure, that is, the probability that a random draw from the first sample (T group) is larger than a random draw from the second sample (V group). The probabilities for FM Quality, Effectiveness and Efficiency were 10.3%, 16.7% and 17.9% respectively. In other words, the rank sum tests for all three samples rejected the null hypothesis meaning that there is significant statistical difference between the group that used text-based tool and the group that used visual SPL tool.

In this experiment, we only evaluated impact of the new tool on novice SPL practitioners when working with relatively small FMs. The experiment results showed several benefits after enhancing the text-based language with FM visualizations. The users not only authored FM of a higher quality but also consistently demonstrated improved productivity expressed in terms of user effectiveness and efficiency. However, further research is required to identify whether this outcome still holds for SPL experts working with much larger FMs.

4.3 Threats to Validity

4.3.1 External Validity

Our evaluation is based on the assumption that we were measuring effects of working with representative FMs that model a real-world artifact. However, the models created as a part of this experiment came from the academic environment, and as such, there is no guarantee that they share characteristics with industrial FMs. Majority of practical FMs have a couple of

hundreds features at most. The number of features we saw in our evaluation FMs ranged from 20s to 70s. One of the largest documented FM in the feature model of a Linux kernel [28] with over 5500 features, and thousands of constraints. While the FM of this scale would clearly pose a challenge to the FAMILIAR Tool in its current state, we will continue to work on improving its performance in the future. Another important aspect to state here is that the experiment population consisted of novice participants exclusively. All of the participants were graduate CS students that had no or very little exposure to SPL and feature modeling prior to this experiment.

Since this experiment was conducted on two geographic locations, and we had limited resources with time constraints, we could not afford to ask both groups to evaluate both tools. Originally, we wanted to use the blocking technique as a part of our experimental design and rotate the groups, asking each group to replicate the experiment with another tool. However, this turned out to be time consuming practice, and we had to adopt more feasible, smaller-scale, option. As a consequence, the Group V, which happen to have somewhat better exposure to SPL and Feature Modeling, served as our treatment group working with visualized FMs. This created the specific situation of the experiment that might limit its generalizability. In order to mitigate this risk, we made sure to provide the same introductory training to all participants.

4.3.2 Internal Validity

When considering the experimental independent variable groups, we used two tools from the same FAMILIAR environment. This helped us to mitigate the tool selection bias risk when evaluating the effect of presenting FMs (e.g., visual vs. textual form). Similarly, all participants were alike (e.g., grad CS students, novice SPL practitioners) with regard to the independent variable.

Repeatedly attempting to perform FM creation during the experiment session, would eventually teach participants to create better FMs in less time. We imposed the restriction of allowing only one session with the FAMILIAR Tool (with no allowed repetitions), regardless of its outcome.

4.3.3 Construct Validity

The most significant threat for the construct validity of the experiment might be due to the fact that all of our dependent variables use an ordinal scale. Does the experimental data provide accurate measurements of what it is intended to measure? According to [39], there are several notable threats caused by the ordinal scale measurement. First, the ordinal labels could be inconsistently interpreted among different users. Second, users might treat ordinal scales as if they had the properties of ratio scales and hence could provide unreliable analysis. Third, the distance between the different labels of an ordinal scale might not present clear comparison between the significance of various ordinal labels. Taken together, these problems could have impacted the construct validity of the FAMILIAR Tool evaluation.

In order to mitigate this threat we used the statistics methods that do respect specifics of ordinal data. Our analysis therefore focused on summarizing the central tendencies and statistical significances rather than trying to come up with exact metrics.

5 Related Work

There are at least few dozen SPL feature modeling tools available today and most of them came from the academic (research) environment during the last decade. Feature modeling tools can nowadays be classified as either graphical or textual, depending on their supported representation of FM's Feature Diagrams. Graphical SPL tools that use FODA [16, 23] notation are most widely used. FeatureIDE [34] is an Eclipse plug-in that provides a comprehensive support for feature modeling, including graphical editors for creating, configuring or reasoning about feature models. In addition, FeatureIDE facilitates the integration with other reasoning tools since it offers a Java API for working with FMs. SPLOT [35] is a web based SPL tool with an interactive FM editor, an automated analysis, a configuration editor with automatic decision propagation and repository of FMs.

However, there are several important differences between those tools and the FAMILIAR Tool. First, to the best of our knowledge, FAMILIAR is the first FM tool with native support for working with multiple feature models. Its composition and decomposition operators can be used to synthesize, refactor, update, compare or reason about FMs. In addition, its multi FM display improves the user understanding of FMs and reduces accidental complexities during the iterative modeling process. Secondly, the FAMILIAR Tool is designed as a standalone rich desktop application. Even though it comes with built-in extensibility and interoperability features, it can function independently of other SPL environments or IDEs. Finally, as far as we know it, this is the first work which completely describes and evaluates the conversion from the text to graphical tooling. Having essentially the same underlying environment for both FAMILIAR tools allows us to observe the potential benefits of FM visualization in isolation, without interfering differences that would likely arise from using various tools.

6 Conclusions and Future Work

The idea behind SPLE is to focus on features that are common and that vary from one product to the other, so as to efficiently produce family of related products by facilitating reuse and adaptation. SPL tools are important part of this discipline as they help automating many operations that could otherwise, if done manually, introduce accidental complexity. As a consequence, this could potentially seriously hinder the quality and efficiency modeling of large scale systems.

Tools improvement continues to represent one of the most important factors critical for the success of a SPL and DSML evolution [7]. In this thesis, our work combines two basic approaches: application-based, which involves developing a new SPL tool for our existing FAMILIAR framework, and experimental, which involves designing a formal experiment on a small-scale to evaluate the usability of this tool and analyzing the experimental data collected. One of the main goals of this thesis was to enhance the text-based FAMILIAR framework by adding graphical support to it. Additional enhancements included supporting FAMILIAR's integrated modeling environment by combining together Configuration Editor, Console interpreter and FM Editor as well as supporting persistence of FMs by implementing FAMILIAR's proprietary file storage given in an XML-based schema. Our motivation was to provide a common environment for modeling practitioners so that they can focus on feature modeling with FODA-like notations without giving up on the interoperability of other FM tools.

As a result of the tool enhancement, we expect to observe several benefits in a future practice. Firstly, in terms of learnability, the learning curve of the FAMILIAR Tool [40, 21] is expected to be favorable for SPL practitioners since new graphical support, when combined together with the restricted set of well-defined operators, provides simple and intuitive learning

concepts. Secondly, in terms of expressiveness, the FAMILIAR Tool completely preserves all of the domain-specific advantages of a DSML. In addition, its existing scripting support adds to this expressiveness. Thirdly, in terms of reusability, the existing modular mechanisms and parameterized scripts, when combined together with FM persistence support, offer modeling solutions that should be readily available for reuse. Fourthly, in terms of interoperability, FAMILIAR's new .treeml file format seamlessly integrates with all of the FM notifications of other SPL tools that FAMILIAR already supports. Finally, in terms of usability, our focus was on user efficiency, effectiveness as well as FM quality. We conducted a small-scale experiment in which we evaluated the impact of model visualizations on the usability in the context of the FAMILIAR environment. For this purpose, we formed two experimental groups from graduate CS students, a treatment group (V-group) that only run new FAMILIAR's visual tool, as well as a control group (T-group) that only run old FAMILIAR's text-based standalone console. Our conclusion was that the new FAMILIAR Tool clearly benefited from FM visualization and helped SPL practitioners to improve their efficiency, effectiveness as well as a quality of FMs they worked with. This was the main goal of the thesis.

In this thesis, after the introduction given in Chapter 1, we introduced basic terms about software product lines, and feature modeling. Chapter 2 provided a background on FM semantics as well as FM manipulations particularly focusing on FAMILIAR's current composition operators. Chapter 3 presented the high-level architecture of the FAMILIAR language. Along this, we also analyzed core aspects of the FAMILIAR framework by exploring different paths in order to enhance it. The overall goal was to take the FAMILIAR platform to whole new level and offer it as a rich and integrated visualized modeling environment. The result of this effort was the newly developed tool. While in the one hand, visual FMs containing a number of

diagrams in various views and at different levels of abstraction are more suitable to the representation and understanding of users' requirements involving various stakeholders, on the other hand, FM textual specifications are more suitable to be used by experienced practitioners for modeling of the more complex workflows or systems. The FAMILIAR Tool helped bridging the gap between those two concepts. We concluded Chapter 3 with sections on two design approaches that we attempted, implementation details as well as use case example of the new tool. Chapter 4 completely described the experiment: its goals, design, hypothesis, methodology, threats to validity, as well as the experimental results with data, statistical analysis and finally, the research conclusion. We needed an in-depth, quantitative evaluation to formally measure the impact of the new tool on practitioner's productivity. However, due to limited people and time resources our experiment was conducted on a rather small population sample.

Future work should be geared towards further improving the FAMILIAR Tool by refining and implementing advanced functional requirements specified in Appendix A as well as evaluating its usability and other relevant metrics in more depth. Another important area where we can further improve FAMILIAR is to further extend its plain FM formalism and take it to the whole new level, that is, support generating configurations or code from FMs. In addition to this, we should attempt to refactor FAMILIAR to make it a generic tool capable of supporting not only software products but families of domain-specific languages. Finally, performing a larger experiment, with the blocking technique as a part of the experimental design and rotating the groups, by asking each group to replicate the experiment with switched tool, would provide us with more comprehensive data on finding a fine balance between visual and text notations in SPL tools, and finding ways to further enhance the existing SPL tooling. This experiment should not necessarily be limited to the FAMILIAR Tool.

REFERENCES

- [1] Acher, Mathieu. "Managing Multiple Feature Models: Foundations, Language and Applications", PhD thesis, 2011.
- [2] Northrop, Linda. "Software product lines essentials." Software Engineering Institute, Carnegie Mel-Ion University (2008).
- [3] Pine, B. Joseph. Mass customization: the new frontier in business competition. Harvard Business Press, 1999.
- [4] Acher, Mathieu, Collet, Philippe, Lahire, Philippe, and France, Robert B. "Managing feature models with familiar: a demonstration of the language and its tool support." Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems. ACM, 2011.
- [5] Acher, Mathieu, Collet, Philippe, Lahire, Philippe, and France, Robert B. "Composing feature models." Software Language Engineering. Springer Berlin Heidelberg, 2010. 62-81.
- [6] France, Robert, and Bernhard Rumpe. "Model-driven development of complex software: A research roadmap." 2007 Future of Software Engineering. IEEE Computer Society, 2007.
- [7] Djebbi, Olfa, Camille Salinesi, and Gauthier Fanmuy. "Industry survey of product lines management tools: Requirements, qualities and open issues." Requirements Engineering Conference, 2007. RE'07. 15th IEEE International. IEEE, 2007.
- [8] Brooks, F. P. "No silver bullet essence and accidents of software engineering. Computer", 20:10–19. 1987.
- [9] Selic, Bran. "The pragmatics of model-driven development." IEEE software 20.5 (2003): 19-25.
- [10] Czarnecki, Krzysztof, and Andrzej Wasowski. "Feature diagrams and logics: There and back again." Software Product Line Conference, 2007. SPLC 2007. 11th International. IEEE, 2007.
- [11] Batory, Don. Feature models, grammars, and propositional formulas. Springer Berlin Heidelberg, 2005.
- [12] Eclipse-Foundation: Xtext, <http://www.eclipse.org/Xtext>.
- [13] Fowler, Martin. Domain-specific languages. Pearson Education, 2010.
- [14] Eclipse-Foundation: EMF, <http://www.eclipse.org/modeling/emf>.

- [15] Acher, Mathieu, Collet, Philippe, Lahire, Philippe, and France, Robert B. "Familiar: A domain-specific language for large scale management of feature models." *Science of Computer Programming* 78.6 (2013): 657-681.
- [16] Kang, Kyo C., et al. "FORM: A feature-; oriented reuse method with domain-specific reference architectures." *Annals of Software Engineering* 5.1 (1998): 143-168.
- [17] Wohlin, Claes, et al. *Experimentation in software engineering*. Springer, 2012.
- [18] Ferber, A., Haag, J. and Savolainen, J: Feature Interaction and Dependencies - Modeling Features for Re-engineering a Legacy Product Line. in Proc. 2nd Software Product Line Conference (SPLC-2), San Diego, CA, USA, (August 19-23 2002). Springer, Lecture Notes in Computer Science, 2002, pp. 235- 256.
- [19] Van Der Storm, Tijs. *Variability and component composition*. Springer Berlin Heidelberg, 2004.
- [20] Johansen, Martin Fagereng, et al. "Generating better partial covering arrays by modeling weights on sub-product lines." *Model Driven Engineering Languages and Systems*. Springer Berlin Heidelberg, 2012. 269-284.
- [21] FAMILIAR Project, <http://familiar-project.github.com>.
- [22] Heidenreich, Florian, et al. "Relating Feature Models to Other Models of a Software Product Line." *Transactions on aspect-oriented software development VII*. Springer Berlin Heidelberg, 2010. 69-114.
- [23] Kang, Kyo C., et al. Feature-oriented domain analysis (FODA) feasibility study. No. CMU/SEI-90-TR-21. Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, 1990.
- [24] Bevan, Nigel. "Measuring usability as quality of use." *Software Quality Journal* 4.2 (1995): 115-130.
- [25] ISO 9241-11. *Ergonomic requirements for office work with visual display terminals (VDTs)-Part 11: Guidance on usability*, 1998.
- [26] Mandl, Heinz, and Joel R. Levin, eds. *Knowledge acquisition from text and pictures*. Elsevier, 1989.
- [27] Winn, William. "Learning from maps and diagrams." *Educational Psychology Review* 3.3 (1991): 211-247.
- [28] Sincero, Julio, and Wolfgang Schröder-Preikschat. "The Linux Kernel Configurator as a Feature Modeling Tool." *SPLC (2)*. 2008.
- [29] Paivio, Allan. "Dual coding theory: Retrospect and current status." *Canadian Journal of Psychology/Revue canadienne de psychologie* 45.3 (1991): 255.

- [30] Heer, Jeffrey, Stuart K. Card, and James A. Landay. "Prefuse: a toolkit for interactive information visualization." Proceedings of the SIGCHI conference on Human factors in computing systems. ACM, 2005.
- [31] Larkin, Jill H., and Herbert A. Simon. "Why a diagram is (sometimes) worth ten thousand words." Cognitive science 11.1 (1987): 65-100.
- [32] Britton, Carol, and Sara Jones. "The untrained eye: how languages for software specification support understanding in untrained users." Human-Computer Interaction 14.1-2 (1999): 191-244.
- [33] Thum, Thomas, Don Batory, and Christian Kastner. "Reasoning about edits to feature models." Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on. IEEE, 2009.
- [34] Christian Kastner, Thomas Thum, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. 2009. FeatureIDE: A tool framework for feature-oriented software development. In Proceedings of the 31st International Conference on Software Engineering (ICSE '09). IEEE Computer Society, Washington, DC, USA, 611-614.
- [35] Kirk, Diana C., Stephen G. MacDonell, and Ewan Tempero. "Modelling software processes: a focus on objectives." Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications. ACM, 2009.
- [36] Van Deursen, Arie, and Paul Klint. "Domain-specific language design requires feature descriptions." Journal of Computing and Information Technology. 2001.
- [37] Classen, Andreas, Quentin Boucher, and Patrick Heymans. "A text-based approach to feature modelling: Syntax and semantics of TVL." Science of Computer Programming 76.12 (2011): 1130-1143.
- [38] Bąk, Kacper, Krzysztof Czarnecki, and Andrzej Wąsowski. "Feature and meta-models in Clafer: mixed, specialized, and coupled." Software Language Engineering. Springer Berlin Heidelberg, 2011. 102-122.
- [39] Hubbard, Douglas, and Dylan Evans. "Problems with scoring methods and ordinal scales in risk assessment." IBM Journal of Research and Development 54.3 (2010): 2-1.
- [40] Jakšić, Aleksandar. "FAMILIAR Tool v1.0.5 (Beta) - Demo."
<http://www.screencast.com/t/BdPgI8yF17Y>
- [41] Czarnecki, Krzysztof, Eisenecker Ulrich W. "Generative programming: methods, tools, and applications". ACM Press/Addison-Wesley Publishing Co., New York, NY, 2000.
- [42] Asikainen, Timo, Tomi Männistö, and Timo Soinen. "Using a configurator for modelling and configuring software product lines based on feature models." Workshop on Software Variability Management for Product Derivation, Software Product Line Conference (SPLC3). 2004.

- [43] García, F. Javier Pérez, Laguna, Miguel A., González-Carvajal, Yania Crespo, and González-Baixauli, Bruno. "Requirements variability support through MDD and graph transformation." Graph and Model Transformation, 2006.

APPENDICES

A. Requirements for FAMILIAR Tool

Core Functional Requirements

- CFR 1. The tool shall support Java standalone GUI application with menus, toolbars, context-sensitive popup menus and an interactive interpreter console.
 - The tool should be delivered as a binary .jar file that will be easy to run. The .jar file will include all of the dependent libraries into a single file.
- CFR 2. The tool shall support visualization of an FM.
- CFR 2a. Visualization Mode: FM in its basic, propositional FODA-Like form with:
 - Mandatory/optional features
 - Feature groups, and
 - Constraints: Implies and excludes relationships
- CFR 2b. FAMILIAR FM operations:
 - Enable building of a FM a tree from scratch:
 - Adding a root feature
 - Adding a child feature (either Mandatory feature or Optional feature)
 - Adding a sibling feature (either Alternative-group, Or-group or And-group)
 - Adding a constraint
 - Altering an FM:
 - Rename a FM name (i.e., a feature model variable)
 - Renaming a feature

- Removing a feature
 - Removing all constraints
 - Reasoning about an FM:
 - Check if a feature model is semantically valid
 - Count features, constraints, valid configurations as well as a depth of an FM tree
 - Display valid configurations, cores and dead feature sets
 - Display a textual notation of a visualized FM
 - Compare FMs
 - Perform checks and enforce rules for
 - syntactic (well-formed rules) checks (i.e., a constraint referring to features that are not in the feature model) and
 - semantic (configuration) checks (i.e., whether a feature model has at least one valid configuration).
- CFR 2c. Visual FM Operations:
 - Expanding/collapsing
 - Zooming in/out
 - Zooming to fit
 - Panning
- CFR 3. The tool shall support FM Configuration operations:
 - Creating a new configuration view
 - Editing a configuration

- CFR 4. The tool shall support multiple views. Use one tab for each feature model or a configuration view:
 - Create a new tab:
 - If an FM or its configuration already exists, a new display is generated from the existing FM/configuration environment variable.
 - If an FM or its configuration does not exist, then a new display is created together with an FM/configuration environment variable.
 - Close an existing tab: Only view of an FM or its configuration is removed. Associated FM/configuration environment variable would continue to exist.
 - Display all of the existing FM/configuration environment variables. See “Create a new tab” from above.
- CFR 5. The tool shall support an embedded FAMILIAR interpreter view: An interactive console that accepts and processes all of the FAMILIAR text-based commands. This will allow creating and/or updating environment variables directly (without graphical edits), as well as using text-based features that have not been yet exposed through the GUI.
- CFR 6. The tool shall support the persistence of FMs (.treeml) by using FAMILIAR’s new proprietary XML file model.
 - Open a FAMILIAR FM
 - Save a FAMILIAR FM
 - Save as a FAMILIAR FM
- CFR 7. The tool shall support interoperability operations with other FM tools/notations through Import/Export menus:
 - SPLOT/SXFM (*.xml)

- FeatureIDE (*.m)
- S2T2 (*.fmprimitives)
- TVL (*.tv1)
- CFR 8. The tool shall support implicit synchronization between two model representations (FM/configuration display as well as environment variables). The third model representation (.treeml) is synchronized (serialize/deserialize) when a user explicitly requests it.
- CFR 9. The tool shall support the help page content.
- CFR 10. The tool shall support enabling/disabling verbose logging.

Advanced Functional Requirements

- AFR 1. Visualization modes: The tool shall support at least two FM visualization modes such as file-explorer like and FODA-like.
- AFR 2. Search Query: The tool shall support a way to navigate in the "feature" space, typically with a search query to look for features names.
- AFR 3. Collapsing: The tool shall support a way "collapse" a sub-tree of an FM.
- AFR 4. Zooming: The tool shall support an advanced zoom technique which relies on the slice operator. When a user zooms on some parts of a feature model, the slicing operator is applied (in the background) by including the features that are currently visible in the editor. This would allow a user to better understand local relations among features.
- AFR 5. Attributes: The tool shall support attributes over features. We need a way to describe a feature by associating information about its rationale, "type", or some qualitative information such as price, performance, etc. This work will also support

editing of attributes, serialization of attributed feature models, specification of attributed feature models, and finally, reasoning over feature attributes.


- AFR 6. Attributes/Colors: The tool shall support additional visual operations such as to colorize features implemented as a special attribute of a feature. This would allow, for example, performing “slicing” of a feature model based on a criterion of the marked features.

Non-Functional Requirements

- NFR 1. Aesthetics: The tool shall support visually appealing and modern look-and-feel GUI.
- NFR 2. Usability: The tool shall support easy to use and intuitive user interface. Modeling practitioners should be able to start using it without additional training time.
- NFR 3. Performance: The tool shall support smooth rendering an FM or configuration tree with over 1,000 nodes (features, groups and/or constraints) on an average PC machine used today.
- NFR 4. Portability: The tool shall run and compile on Windows, Mac and Linux platforms.
- NFR 5. Modifiability: The tool code shall be object-oriented, well-designed, extendible, and maintainable.
- NFR 6. Platform Constraints: The tool shall be backward compatible with Java Runtime/SDK 5, 6 and 7. In addition, the tool shall run on all supported platforms (see NFR 4) independently of the Eclipse environment.

B. Evaluation Scenarios

We initially planned to carry out six Tasks, but since we had time and resource restrictions, we scaled the experiment down to the Task0 and Task1 only which ask participants to do the modeling work with the FAMILIAR Tool by creating a new feature model of the online AUDI configurator. For the sake of completeness we present all of the original tasks:

Task	Description
Task0	<p>Visit the AUDI configurator and spend some time (~10 minutes) to play with the configurator and observe how it works. You will only do this task one time for the whole experiment, however don't forget to record your actions.</p>  <p>The screenshot shows the Audi configurator interface. At the top, there's a grid of car models with labels: A1, A3, A4, A5, A6, A7, A8, Q3, Q5, Q7, TT, R8. Below the grid is a configuration panel with three main sections: Model line, Body style, and Model. Each section contains a list of options with radio buttons. At the bottom, there's a progress bar with six steps: 1 Model, 2 Engine, 3 Exterior, 4 Interior, 5 Equipment, 6 Your Audi. Step 1 is currently selected.</p>
Task1	In 20 minutes, create a partial feature model supporting the whole structure of the AUDI configurator.
Task2	Take a “model line” (e.g. Audi A1, Audi A3, etc.) and create a feature model representing all the variability of the model line. Follow only the step 1 and 2: Exterior, Interior and Equipment are not required. Try to go to the essential and don't spend too much time on details. Also, don't forget that FAMILIAR only

	allows strings in feature names.
Task3	<p>Let us play with our feature model. First, let us check some (basic) properties and better understand our specification:</p> <ul style="list-style-type: none"> • the feature model represents at least one valid configuration • there is no “dead” feature • features included in all configurations (core features) of the feature model should correspond to non-configurable options (e.g. steps, category, containers’ name) • there is no “false optional” feature • create some partial configurations that are actually consistent with the behavior of the configurator • create some complete configurations that are actually consistent with the behavior of the configurator
Task4	Repeat tasks 1 or 2 for each model line. Name your different files with the model name (e.g. Task1_AudiA1.fml, Task1_AudiA1.ovg, Task2_AudiA1.fml, etc.)
Task5	When you finished all feature models, merge it into a single one and test the same properties of task 2. Compare the obtained feature model with the feature model of task 1. Put your conclusions in few lines in a text file.
Task6 (bonus)	As the number of configuration options can be huge, we decide to build a set of simplified feature models in order to focus only on some aspects of an Audi car. Propose and try different strategies using decomposition mechanisms.