

UTFM – a Next Generation Language and Tool for Feature Modeling

by

Vincent Weber, BSc

Thesis

Presented to the Faculty of Electrical Engineering, Mathematics and Computer Science of

the University of Twente

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

University of Twente

August 2014

Copyright

by

Vincent Weber

2014

The Supervising Committee for Vincent Weber
certifies that this is the approved version of the following thesis:

**UTFM – a Next Generation Language and Tool for Feature
Modeling**

Committee:

Dr. Pim van den Broek, Supervisor

Prof. dr. Mehmet Aksit

Prof. dr. Don Batory, External Supervisor

Acknowledgments

Before we delve into the world of feature modeling, there is the moment to reflex and appreciate the people that made my work, this thesis, a masters project, possible. As one will see in the thesis, structure and hierarchy are key, therefore I would like to begin with the persons that had a direct influence on my results.

First, I'd like to thank Mehmet for introducing me to variability and feature modeling through his courses, and thereafter helping me find a research position abroad. Within six week I knew where I was heading, mainly due to the contacts he put me in touch with. Secondly, thank you Don! For being the host for my research internship at the Computer Science department of the UT at Austin (TX). From the start I have felt welcome and I am grateful for the freedom that has been given for my research. When I needed guidance or feedback there was always some free space in his agenda. It were eight pleasant months working at his office, during which I've learned so much. Last, I'd like to thank Pim for all his efforts to push me to get the best results, furthermore correcting (and parsing, and typechecking) my thesis to the smallest detail. Without, the result would not have been the same.

The University of Texas at Austin, and the city itself, has made a great impact. I would like to thank the institute, with its staff and students for the wonderful experience, through which I bleed a little burned orange nowadays. Hook 'em!

Besides during my research, I have always been supported by family and friends. Fred, Ludy en Martijn thank you for always being there, when I wanted to share my

accomplishments or needed moral support to continue. With this thesis my time as a student ends, and for this unforgettable time I'd like to thank my friends from my yearclub, my fraternity Ius Sanctus and my house Kroegzicht, and all other close friends.

VINCENT WEBER

University of Twente

August 2014

UTFM – a Next Generation Language and Tool for Feature Modeling

Vincent Weber, MSc
University of Twente, 2014

Supervisor: Dr. Pim van den Broek

An important aspect of *variability management* in *software product lines* is through *feature modeling*, in which relations between different features are specified. Over the years extensions of classical feature models have been proposed, however never properly combined and implemented. We propose the next generation feature modeling language: *UTFM* (University of Twente/University of Texas Feature Models), that entail generalized classical feature models with instance and group cardinalities, extended with *feature replication*, *feature attributes* (arithmetic, boolean and string attributes) and *complex cross-tree constraints*.

The language explicitly separates *type declarations*, defining feature types, hierarchical relations, attributes and constraints, from model configurations. *Configurations* consist of instance declarations of feature types and value assignments to feature attributes. We explain our interpretation of the semantics of *nested feature replication*. We introduce a *local scope* to features and explain how such a scope influences the constraint language used

in cross-tree constraints, as well as how this leads to a language that has similar behavior as an *attribute grammar*. Along we propose the automated analysis operation *constraint propagation*. An algorithm for propagating determinable values to instances and attributes based on provided configurations. The propagation algorithm is supported by an *unfolding* process that makes undecided instances in a configuration explicit.

To validate the proposed language and analysis operations an proof-of-concept tool is implemented. In order to check satisfiability of configurations the tool translates UTFM models to Z3 decision problems (an off-the-shelf SMT theorem prover). To translate the models to the decision problems an UTFM to Z3 mapping is described for the various semantics. Furthermore two examples are provided that show case the capabilities of UTFM.

As a whole, the UTFM language and tool, set a step forward to converge feature models with advanced semantics towards standards, furthermore to enrich variability modeling using feature models.

Contents

Acknowledgments	iv
Abstract	vi
List of Figures	xi
Chapter 1 Introduction	1
1.1 Next generation feature modeling	2
1.2 Motivation	3
1.3 Problem Statement	3
1.4 Research Goals	4
1.5 Methodology	5
1.6 Contributions	5
1.7 Outline of Thesis	6
Chapter 2 Classical Feature Models	7
2.1 Classical Semantics	7
2.2 UTFM Type Declarations and Configurations	10
2.2.1 Type Declaration	11
2.2.2 Constraint Language	12
2.2.3 Configurations	15

2.3	Mapping to UTFM	18
2.4	Automated Analysis with Z3	19
2.4.1	Translating to Z3	21
2.4.2	Automated Analysis	24
Chapter 3	Feature Replication	31
3.1	Replication of Features	31
3.2	Multiple Instances in UTFM	33
3.2.1	Type Declaration	33
3.2.2	Constraint Language	34
3.2.3	Configuration	36
3.3	Automated Analysis with Z3	38
3.3.1	Translating to Z3	38
3.3.2	Constraint Propagation	39
Chapter 4	Attributed Feature Models	41
4.1	Classical Attributed Feature Models	41
4.2	Attributes in UTFM	43
4.2.1	Type Declaration Extensions for Attributes	43
4.2.2	Constraint Language Extensions for Attributes	44
4.2.3	Configurations with Attributes	51
4.2.4	Mapping Classical Attributed Feature Models to UTFM	52
4.3	Automated Analysis with Z3	53
4.3.1	Mapping to Z3	53
4.3.2	Constraint Propagation	55
Chapter 5	UTFM Tool	57
5.1	Functionality and Tool usage	57
5.2	Implementation	62

Chapter 6	Examples	65
6.1	Mobile Device Interface SPL	65
6.1.1	Entities	66
6.1.2	Navigation	67
6.1.3	Window Size	69
6.1.4	Stylesheet	71
6.1.5	Tool execution	73
6.2	Subsea Oil Production SPL	73
Chapter 7	Related Work	78
Chapter 8	Conclusion and Future Work	82
8.1	Results and Conclusion	82
8.2	Future Work	84
8.2.1	Reuse and Modularization	84
8.2.2	Optimization	84
8.2.3	Integer Programming	85
8.2.4	Front-end	85
Appendix A	Context-free Grammars UTFM	87
A.1	Type declaration	88
A.2	Configuration	89
A.3	Product	90
A.4	Constraint Language	91
Appendix B	Tool Execution Options	93
Bibliography		94

List of Figures

1.1	Classical feature model of a Car [7].	2
2.1	Classical Feature Models	10
2.2	Feature model type declaration of the GraphLibrary example (Figure 2.1). .	13
2.3	Context-free grammar or UTFM constraint language for classical models. .	14
2.4	Simple example of <i>exists</i> operations.	14
2.5	More complicated constraints.	15
2.6	FMTD of Figure 2.2.	16
2.7	Configuration of Figure 2.6 in UTFM.	17
2.8	Graphical representation of Figure 2.7.	17
2.9	Very basic Z3 satisfiability problem.	20
2.10	<i>Cardinality</i> function.	20
2.11	Instance and instance cardinality in Z3.	22
2.12	Group and group cardinality in Z3.	22
2.13	Cross-tree constraint in Z3.	23
2.14	Automated analysis process.	24
2.15	Configuration of Figure 2.7.	26
2.16	Propagated configuration of Figure 2.15.	26
2.17	Unfolded configuration of Figure 2.16.	29
2.18	Full configuration of Figure 2.17.	30

2.19	Product specification of Figure 2.18.	30
3.1	Feature model with replication from [15].	32
3.2	Classification of different cardinality interpretations.	34
3.3	<i>Forall</i> operation examples.	36
3.4	Type declarations for Figure 3.3.	36
3.5	Multiple configurations of Figure 3.3b.	37
3.6	Z3 translation of an <i>exists</i> operation on a non singular feature type.	38
3.7	Z3 translation of cross-tree constraint of Figure 3.3b using Figure 3.5d . . .	39
3.8		40
4.1	Attributed feature model by Benavides et al.[7].	42
4.2	Part of the graphical FTMD representation of Service feature model (Figure 4.1).	45
4.3	Part of the FMTD of Service feature model (Figure 4.1).	45
4.4	Constraint language grammar of Figure 2.3 extended to support attributes. .	49
4.5	Substitution example.	51
4.6	Example configuration of Figure 4.3.	52
4.7	Z3 arithmetic attribute.	53
4.8	Z3 boolean attribute.	54
4.9	Z3 arithmetic constraint.	54
4.10	Z3 maximum and minimum functions.	55
4.11	Z3 maximum and average example.	55
4.12	Configuration after constraint propagation of Figure 4.6.	56
5.1	Overview on analysis processes in the tool	59
5.2	Validating type declarations.	59
5.3	Validating configurations.	60
5.4	Validating products.	61

5.5	Translating configurations to products.	61
5.6	Propagating constraints for configurations.	62
5.7	Unfolding and propagating constraints for configurations.	62
5.8	Internal structure of the UTFM tool	63
6.1	Window FMTD.	68
6.2		68
6.3	Window navigation constraints.	69
6.4	Possible scrollability constraints.	69
6.5	Window size constraints.	70
6.6	Complex constraints to constraint the window size.	70
6.7		71
6.8	Stylesheet configurations of Figure 6.7a.	72
6.9	Stylesheet configuration of Figure 6.7b.	73
6.10	The subsea oil production system fragment from [6].	74
6.11	UTFM representation of the subsea oil production SPL fragment.	74
6.12	FMTD translation of SimPL fragment in Figure 6.11.	75
6.13	Configuration of SimPL fragment Figure 6.10.	77
6.14	Configuration translation of SimPL configuration in Figure 6.13.	77
A.1	Context-free grammar of FMTD's.	88
A.2	Context-free grammar of Configurations.	89
A.3	Context-free grammar of Products.	90
A.4	Context-free grammar of the Constraint Language.	92

Chapter 1

Introduction

Producing individual tailored software products efficiently is known in software engineering as *software product lines* (SPL) or *software product families*. Customizing reusable artifacts is the key, where the focus is on *features* functionalities that are shared among members. The *software product line engineering* paradigm models and manages variability, i.e. commonalities and differences in the applications in terms of requirements, architecture, components and test artifacts [27].

Kang et al. [23] introduced feature modeling in their Feature Oriented Domain Analysis (FODA) publication. *Feature models* (FM) represent the relations between the features of a product line in a hierarchical way. Legal combinations of features are in 1:1 correspondence with products of a product line, thereby describing an entire family of products in a single model. Figure 1.1, published by Benavides et al. [7], illustrates a basic example of a feature model describing various mobile phones, with variability in screens and media features. As explained in the legend of the figure, certain features are mandatory, while others are optional, and some features are dependent of another, and some are mutual exclusive. By selecting features a configuration is made of the feature model.

The information in a FM can be used in various ways, and the process of extracting and analyzing a model or configuration can be automated with the use of tools. Especially in

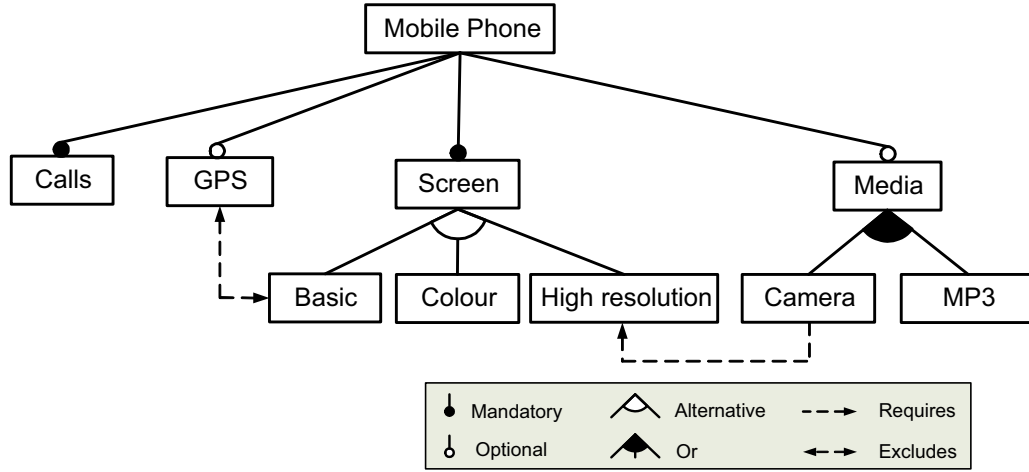


Figure 1.1: Classical feature model of a Car [7].

large-scale feature models automated analyses is unavoidable as manually analysis will be an infeasible task due to the large computing complexity of the analysis [5]. The automated analysis operations performed by tools can vary from detecting contradictions in a model, searching for all possible products, and providing optimizations of a model [7].

The operations to perform the analysis all follow the same process, a feature model is used as input, either with a configuration, and is processed by a compiler. The intermediate representation that is returned by the compiler is then passed on to an off-the-shelf-solver or a tool which performs the analysis operation and returns the result of the analysis. The intermediate representations of the feature models are in general satisfiability problems, which the solvers try to validate. Representations that have been used in the past are propositional logic [], constraint programming [], linear programming [] and generalized feature trees [].

1.1 Next generation feature modeling

In the domain of feature modeling a distinction can be made between classical feature modeling and a newer, next generation, feature modeling. The semantics of classical feature modeling are fairly basic as can be seen in Figure 1.1, however over the years researchers

started adding extra semantics to feature models such as cardinalities, feature attributes and allow feature replication (cloning). These new semantics have not matured to the point that they also regarded as classical semantics.

An attempt has been made recently to unite many concepts for a the next generation feature models in Common Variability Language [], however unfortunately this proposal has been stranded (i.e. discontinued as the basis of an OMG standard). This attempt shows us that there is interest in uniting the parallel researches to work towards more standards in feature modeling. In our work we attempt to do provide a full implementation and description of a next generation feature modeling language, *University Twente/University of Texas Feature Modeling* (UTFM).

1.2 Motivation

To date much is unclear on how to interpret the advanced semantics of feature models and how they should be properly integrated with each other. Better insight in the possibilities that richer semantics bring leads to a better understanding of the effects of richer semantics on the structure of a feature modeling language. The implementation of languages could lead to new standards in feature modeling, which would make it more likely that better tools will be developed as the research field becomes less experimental. In order to proceed in such a direction we propose the next generation feature modeling language UTFM and implement a proof-of-concept tool for our proposed language and analysis operations.

1.3 Problem Statement

Defining a feature modeling language with advanced functionality poses two main challenges. The first challenge comes forth from the absence of standards in feature modeling, the second from the interference between the different advanced semantics.

Research of feature modeling has not diverged to standards, on the contrary recent

publications has branched and made the research field more fragmented. Due to the different publications on additional semantics for classical feature modeling it is unclear: 1. to which extend is it possible to add additional functionality; 2. what should this language look like; 3. to what kind of satisfiability problem should a feature model be translated; 4. what are practical examples that can be used to showcase the functionality of the language. The different interpretations of feature attributes and feature replication have been published using different syntax, makes it hard to compare the proposals. Besides the lack of consensus on what feature modeling language with advanced semantics should entail the translation from model to satisfiability problem adds more complexity to the equation. The spectrum of interpretations has also not lead to model examples that are used as standard examples through different publications which would make comparison of interpretations easier.

The versatility has not lead to a feature modeling language that implements advanced feature modeling functionality in a single language. Attempts to unify previous proposals of advanced semantics have not been finalized. The absence of successful proposals for advanced feature modeling languages leads to the suggestion that the combinations of attribute and replication semantics to implementations challenges. UTFM is required to integrate these semantics without interfering with each other. The challenge designing UTFM is to sort out these differences and make design decisions and implement a proof-of-concept tool accordingly.

1.4 Research Goals

The main goal of our work is to develop a full implementation of the feature modeling language UTFM that bundles previously proposed advanced feature modeling semantics besides the classical semantics. Our point of reference is classical feature modeling along with advanced semantical concepts that have been published. In order to incrementally advance in going from our point of reference to the proposed research goal we define the following three sub-goals:

1. define the semantics and syntax of UTFM;
2. implement a parser and automated analysis tool for the UTFM language;
3. provide examples that demonstrates the capabilities of the functionality of the language.

1.5 Methodology

First through a process of trail-and-error an overview is made on the semantics of the designed language, how the different advanced semantics of feature modeling can be combined without interfering with other semantics. Through this process a series of design decisions are made which lead to the structure of UTFM. By defining a context-free grammar the syntax of the language is specified.

Secondly by developing a tool which automates the validation of constructed feature models in UTFM, and implements automated reasoning operations shows the feasibility of the previously stated modeling language. Additionally, by performing automated reasoning, it demonstrates that current off-the-shelf theorem provers are capable of analysing advanced semantics of feature models.

For the third sub-goal examples demonstrate possible usage of UTFM language combining advanced feature modeling semantics in a practical problem domain.

1.6 Contributions

Within this thesis the feature modeling language UTFM is presented that implements feature attribute, feature replication and complex cross-tree constraint semantics. For the language a textual and graphical syntax are provided. Furthermore a tool is implemented that performs automated analysis on feature models written in UTFM using the off-the-shelf theorem

prover Z3¹.

1.7 Outline of Thesis

Chapter 2 starts of with the introduction to the semantics and syntax of UTFM, which are described by providing a mapping from classical feature models to the proposed language. Furthermore it elaborates on the automatic reasoning process: constraint propagating and the supporting unfolding algorithm.

Chapter 3 discusses how UTFM supports the replication of features and how feature models can be constraint when more then one instances are present in a product configuration. Along it is explained how multiple instances, and cardinalities of features are interpreted and how it impacts automated reasoning.

In Chapter 4 the last semantics of UTFM are explained with the introduction of feature attributes. It is explained how attributes are declared for features and instances and how these attributes constraint configurations. This is followed by a description on the automated analysis of features that have attributes.

The proof-of-concept tool is introduced in Chapter 5. The functionality of tool as well as how to execute these operations are described. Furthermore the implementation structure of the tool is elaborated.

Chapter 6 provides examples that showcase the potential of the UTFM language.

Chapter 7, the related work section describes the latest research that is published with regard to feature modeling with advanced semantics and automated analysis. These works are placed in context with our proposed language.

Chapter 8 concludes the thesis by providing the final discussion and summary of the contributions that are made. In the end recommendations are made on the direction of future research and possible enhancements of UTFM language and tool.

¹<https://github.com/vweber/UTFM>

Chapter 2

Classical Feature Models

In this chapter we explain classical feature modeling in UTFM language.

1. We review ideas originally proposed by Kang in 1990 and those added by others through 2006, that we regard as classical feature modeling;
2. We explain the structure of how models in UTFM are declared and configured;
3. A set of translation rules is given to map classical feature models to UTFM, showing that UTFM preserves classical capabilities;
4. The theorem prover Z3 is introduced along with rules to translate UTFM specifications to Z3 specifications. Z3 performs automated reasoning on UTFM feature models;

These topics provide insights into the fundamental concepts of UTFM, which are used in future chapters to introduce new (or more precise) semantics in UTFM feature models and automated reasoning.

2.1 Classical Semantics

Feature Models. In 1990, Kang et al. [23] introduced feature models to encode the program membership of SPLs. His ideas were gradually refined by others (Benavides et al. [10],

Czarnecki et al. [15] and Batory et al. [4]) to what we now regard today as classical feature models. These models deal with three central concepts: *features*, *hierarchical structure*, and *cross-tree constraints*. We describe how classical feature models are interpreted in the past.

Features. A *feature* is an increment in functionality. A feature model enumerates the features used in a software product line. Every program in an SPL is identified by a unique set of features. This set of features is called a *configuration* and each feature in the set is said to be *selected*.

Hierarchical Structure. Features are related to each other by a tree structure, starting with a root feature and descending downward using parent-child relations to produce a hierarchy. A key rule of configurations is that if a child feature is selected, its parent feature must also be selected.

The children of a parent form a *group* relation w.r.t. the parent. There are three different groups: *or*, *and*, and *alternative* relations. An *or* relation requires one or more children to be selected in a configuration. An *alternative* relation requires precisely one child to be selected, and an *and* relation requires all children to be selected (with the exception of children that are labeled as *optional*).

Czarnecki et al. [14] generalized the *group relation* by adding cardinalities, whose syntax is similar to multiplicities in class diagrams of UML. A *cardinality* is an ordered pair $[1..u]$ of integers, where $1 \leq u$. Value 1 is the lower bound on the number of children that must be selected in a group and u is the upper bound, this is called the group cardinality of a relation. Besides cardinality for group relations, Czarnecki et al. [15] introduces feature cardinality, to specify the amount of allowed clones of a feature. Assuming classical feature models do not entail feature replication, there are only *optional* features ($[0..1]$ feature cardinality) and *mandatory* features ($[1..1]$ feature cardinality).

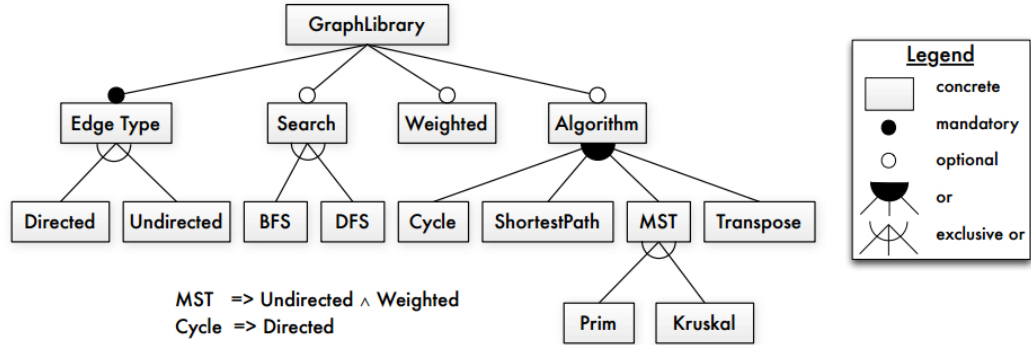
We interpret the cardinalities 1 and u with regard to the different group relations

as follows: suppose a parent node has n children, of which k are optional. An *or* relationship is indicated by the cardinality $[1..n]$, an *and* relationship has the cardinality $[(n - k)..n]$ (where $n - k$ are the mandatory features), and an *alternative* relationship is $[1..1]$, where all child features in the group relation have a feature cardinality (defining whether the feature is optional or mandatory).

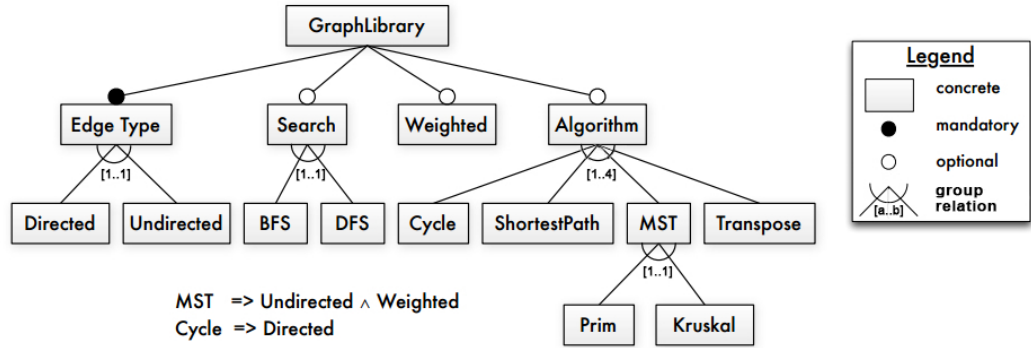
Cross-Tree Constraints. Not all relationships among features can be expressed as parent-child. Features that are selected in one branch of a tree may preclude or demand the selection of features in other branches. *Cross-tree constraints* express such relationships. At first cross-tree constraints entailed *requires* and *excludes* predicates that related two features. Batory [4] proposed that cross-tree constraints can be arbitrary propositional formulas. Some tools today still only allow *requires* and *excludes* constraints, e.g., KConfig [34, 37].

Figure 2.1a illustrates a classical feature model, provided by Apel et al. [1]. This model uses different group relations and is constrained by two propositional formulas. In the Figure legend, *exclusive or* relations are identical to the *alternative* relations described earlier. Figure 2.1b illustrates the same feature model using group relations and cardinalities. This figure is not provided by Apel, but created by us.

Configurations. Products of a product line are specified by configurations of a feature model. We mentioned earlier that a configuration is a subset of features that have been selected. The configuration represents a product, or products, in which those features are present. Benavides et al. [7] formalizes it as: there is a set of features F and a 2-tuple configuration (S, R) , where S the set of selected features, R the set of removed features such that $S, R \subseteq F$, and $S \cap R = \emptyset$. In a *full* or *complete configuration* every feature of the model is either selected or removed ($S \cup R = F$); it is a *partial configuration* otherwise ($S \cup R \subset F$). Batory [4] said features were *undecided* when features are neither selected or removed. We regard this as $S \cup R \cup U = F$, where U is the set of *undecided* features. A *valid full* configuration requires



(a) Classical feature model from Apel et al. [1].



(b) Modified with group cardinality.

Figure 2.1: Classical Feature Models

that its selected features satisfy all feature model constraints; the full configuration is *invalid* otherwise. A partial configuration is *valid* if it can be configured further (by converting *undecided* features into selected or removed) to produce a valid full configuration. A partial configuration is *invalid* otherwise.

2.2 UTFM Type Declarations and Configurations

Classical feature models and their configurations distinguished (albeit implicitly) between a feature type declaration (the feature model) and feature type instances (configurations). With the introduction of UTFM we make the same distinction, but explicitly, as we separate their

concerns. The *feature model type declaration (FMTD)* of UTFM subsumes (and as we will see later chapters, goes beyond) the semantics of classical feature models. In the following sections, we introduce the UTFM constraint language, and then UTFM configurations. We limit our discussion of UTFM concepts to just describe classical semantics. Therefore we make assumptions, which in the future we will retract to achieve generality.

2.2.1 Type Declaration

A FMTD specifies the structure and constraints of a UTFM.

Features. A “feature” in a FMTD denotes a *feature type*; it has no configuration information.

Instances of a feature type are actual features in a product, sometimes called *feature replicas*. Classical feature models use *singleton feature types*: types that have precisely one instance. (It was for this reason that feature types and feature instances were not clearly distinguished in classical models).

More generally, each feature type in a FMTD has an *instance cardinality* that defines the lower and upper bound on the number of its instances that may be permitted in a product. Using the cardinality notation defined earlier ($[1..u]$ means 1 is a lower bound on the number of instances and u is the upper bound), a *mandatory* feature has the cardinality $[1..u]$ and an *optional* feature has $[0..u]$.¹ In classical models, instance cardinalities are either *mandatory* or *optional*, and under the assumption that there can only be a single instance of a type, the upper bound is 1.

In classical feature modeling, a feature A meant “ A ” is the name of the feature type and “ A ” is the name of its sole instance. However in UTFM, with the possibility of feature replication, this means A (a non-indexed term) denotes a feature type and A_i (an indexed term) denote the i th instance of A . All instances of a type have unique names via their index values.²

¹When instance cardinality was introduced, it was decided to leave out redundant syntax [14]. For UTFM, we decided to always express instance and group cardinalities, consistency is valued over conciseness.

²UTFM and the tool do not require the A_i name convention for instances; any name is permitted for an

Hierarchical Structure. All group relations (*and*-, *or*-, *alternative*-) are replaced by a general group relation with a $[1..u]$ group cardinality. This means that both feature types *and* group relations have cardinality specifications in a FMTD. A group relation is part of the declaration of a parent feature type and is associated with one or more child feature types.

CrossTree Constraints. Cross-tree constraints expressed in propositional logic, not limited to elementary *requires* and *excludes* statements. They are declared within the local scope of feature types and therefore are no longer regarded as global constraints. This restriction makes the translation of classical cross-tree constraints to FMTD more elaborate; the benefits for doing so are explained in later sections. Figure 2.2 illustrates how a constraint belongs to a feature type, annotated with a dashed line to the owner. The constraint language of UTFM and possible operations are explained in Section 2.2.2.

Figure 2.2 shows the FMTD representation of the classical feature model of Figure 2.1b. All group relations have been replaced with group cardinalities and each feature type has an instance cardinality. Every instance cardinality is mandatory $[1..1]$ or optional $[0..1]$ in relation to its parent. Cross-tree constraints are specified in a parallelogram, within the scope of the root feature. Also our notation is similar to that of the *Common Variability Modeling (CVL)* [25] proposal for cross-tree constraints.

2.2.2 Constraint Language

Cross-tree constraints written in UTFM are boolean expressions that are required to hold within the scope of a feature type. Expressions can either be a propositional formula or a boolean operation on a feature type. Propositional logic constructs included in the constraint

instance. It is not even required to specify global unique names for instances, just unique names for instances within a group relation. The unique identifier for an instance is its path in the configuration. For the sake of clarity and to avoid ambiguous examples, every instance in our examples have unique names to show explicitly that every instance is unique.

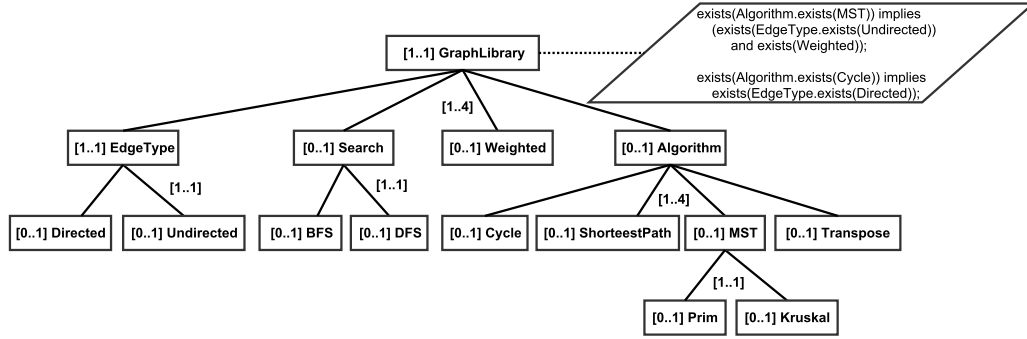


Figure 2.2: Feature model type declaration of the GraphLibrary example (Figure 2.1).

language are: *and*, *or*, *implies*, and *not*. As we will ultimately deal with sets of feature type instances, we reference instances via operation calls. For now we only introduce a single boolean operation for types: *exists*.³ Figure 2.3 states the context-free grammar for UTFM constraint language for classical feature models. A complete grammar of the constraint language is provided in Appendix A.4.

Exists(FeatureType) :: Bool. Operation *exists* returns a boolean value indicating if there exists a selected instance of the specified type: *true* if there is at least one selected instance of the feature type in the configuration, otherwise *false*.⁴

Let parent P denote a feature type for which a cross-tree constraint has been defined, and let C be a child of P . The expression $\text{exists}(C)$ defined at P returns *true* if there is at least one child instance C for the parent instance of type P that is selected in the configuration, otherwise *false*. If the P instance has multiple child instances of C selected *exists* also returns *true*, this is however outside the scope of classical feature models.

As a concrete example, Figure 2.4 declares that each parent P instance requires at least one A child *or* at least one B child instance. Constraints on the selection of Child

³There is also a *forall* operation which is introduced later in Chapter 3. At this point, *exists* and *forall* are semantically indistinguishable as feature types have at most one instance.

⁴UTFM permits types with no instances, or instances with a *false* or *undecided* selection value. This is discussed in the next subsection on UTFM configurations.

```

1 <Constraint> ::= <BooleanExpr> ";"
2
3 <BooleanExpr> ::= <BooleanFormula>
4
5 <BooleanFormula> ::= <BooleanProp> "and" <BooleanProp>
6   | <BooleanProp> "or" <BooleanProp>
7   | <BooleanProp> "implies" <BooleanProp>
8   | "not" <BooleanProp>
9
10 <BooleanProp> ::= <BooleanOp>
11
12 <BooleanOp> ::= "exists(" <FeatureRef> ")"
13   | "exists(" <FeatureRef> "." <BooleanOp> ")"
14   | "forall(" <FeatureRef> ")"
15   | "forall(" <FeatureRef> "." <BooleanOp> ")"
16
17 <FeatureRef> ::= <String>

```

Figure 2.3: Context-free grammar of UTFM constraint language for classical models.

instances are defined as constraints at the feature type that has the `Child` type in its group relation.

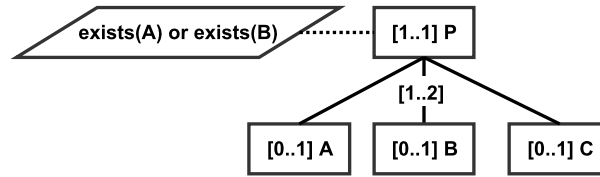


Figure 2.4: Simple example of *exists* operations.

Operations on feature types can be nested. Figure 2.2 illustrates a more complicated constraint (reproduced in Figure 2.5) with nested *exists* operations. The nested operation `exists(Algorithm.exists(MST))` asks if a `GraphLibrary` instance has a child `Algorithm` instance that has a child `MST` instance – or more compactly, if the `GraphLibrary` instance has a selected `MST` instance. The explanation of the implicitly-conjoined constraints in Figure 2.5 is:

- If an instance of `MST` is selected then selected instances of `Undirected` and `Weighted`

are required for the first expression to be *true*.

- If an instance of `Cycle` is selected, it is required that an instance of `Directed` is also selected.

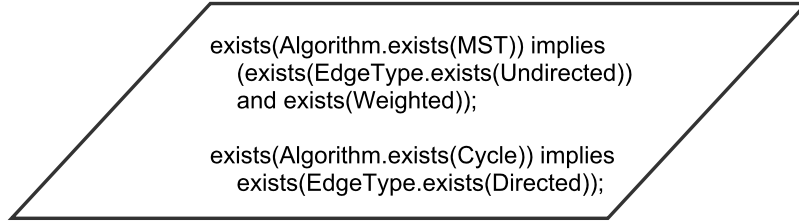


Figure 2.5: More complicated constraints.

A complete textual specification of Figure 2.2 in the UTFM language is given in Figure 2.6. In the textual language curly braces are added to avoid ambiguity in the hierarchy of the model. The mapping of classical feature models to UTFM is explained in the next section. A context-free grammar for specifying FMTD's is proved in Appendix A.1.

2.2.3 Configurations

A FMTD represents the set of all legal products for an SPL. Specifying a product is the process of *configuration*. A configuration consists of instances of feature types. Every instance has a selection value that is either *true* if the feature instance is present in a configuration, *false* if the instance is not. The default selection value of an instance is *undecided*. If the selection value is set to *true* we say the instance is *selected*, if the selection value is not *undecided* anymore an instance is configured. As the default selection value of instances is *undecided*, omitting the specification of *undecided* instances is allowed.

We require that every feature instance be given a distinct name within a group relation. As mentioned earlier, instance i for feature type A has name A_i , although our language and tool admits non-numbered names. So given feature types parent P , child C and grandchild G , the unique path for grandchildren instances would be $P_i.C_j.G_k$.


```

1  [1..1] GraphLibrary
2    constraints:
3      exists(Algorithm.exists(MST)) implies exists(EdgeType.exists(
4        Undirected)) and exists(Weighted);
5      exists(Algorithm.exists(Cycle)) implies exists(EdgeType.exists(
6        Directed));
7    group [1..4]:
8      [1..1] EdgeType
9        group [0..1]: {
10          [0..1] Directed
11          [0..1] Undirected
12        }
13      [0..1] Search
14        group [0..1]: {
15          [0..1] BFS
16          [0..1] DFS
17        }
18      [0..1] Weighted
19      [0..1] Algorithm
20        group [1..4]:
21          [0..1] Cycle
22          [0..1] ShortestPath
23          [0..1] MST
24          group [1..1]: {
25            [0..1] Prim
26            [0..1] Kruskal
27          }
28          [0..1] Transpose

```

Figure 2.6: FMTD of Figure 2.2.

Instance declarations have the following syntax: `(undecided) GraphLibrary : GraphLibrary1`. An instance of the type `GraphLibrary` is declared with the name `GraphLibrary1` and has an *undecided* selection value that needs to be configured. The selection value could be replaced by `(true)` or `(false)`, however to make the language more concise it is allowed to omit the selection value for *selected* instances. A configuration of Figure 2.2 is given in Figure 2.7, also a graphical representation of the configuration is provided in Figure 2.8.

The classifications that were made for configurations of classical feature modeling can also be defined for UTFM. Configurations are either *full* or *partial* and are *valid* or *invalid* with regard to its type declaration.

```

1  GraphLibrary : GraphLibrary1
2    group:
3      EdgeType : EdgeType1
4      group: {
5        Undirected : Undirected1
6      }
7      Weighted : Weighted1
8      Algorithm : Algorithm1
9      group:
10       MST : MST1
11       group:
12         Prim : Prim1

```

Figure 2.7: Configuration of Figure 2.6 in UTFM.

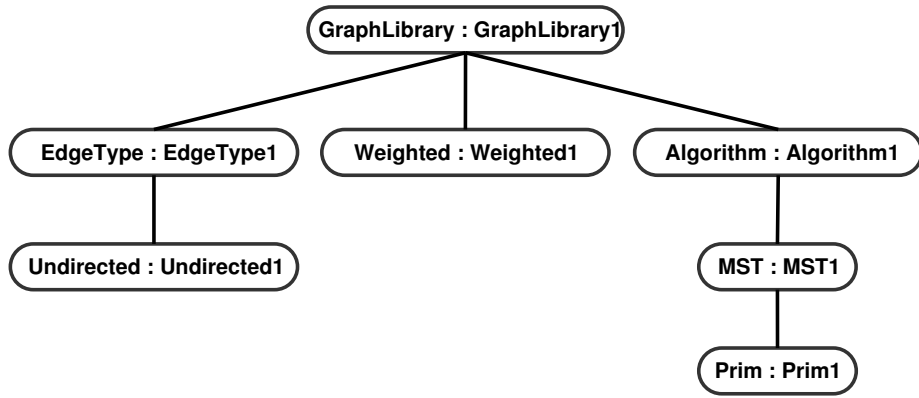


Figure 2.8: Graphical representation of Figure 2.7.

Full configuration. A *full* configuration represents a single product, implying that there are no unconfigured instances in the configuration (no *undecided*).

Partial configuration. On the other hand in a *partial* configuration there are *undecided* instances, and the configuration describes one or more products.

Valid configuration. If a configuration is *valid*, all instance selections and removals do not invalidate the UTFM.

Invalid configuration. Instance selections and removals violate the constraints of the UTFM.

While by default instances are *undecided*, we are unable to specify a full configurations, because implicitly there are always unconfigured instances. All configurations in UTFM are therefore partial configurations. From this point on full configurations will be called products, a configuration of a FMTD in which only selected instances are specified. By default everything else in a product is assumed *false*, and is omitted from the product specification. A similar approach is described in the configuration process for GUIDSL [4].

The configuration presented in Figure 2.7 and Figure 2.8 is *partial* and *valid*. If we convert the domain from configuration to product in the example, the figure represents a *valid* and *full* configuration, in other words a *valid* product. The context-free grammar for specifying configurations is provided in Appendix A.2, for products in Appendix A.3. The difference between these grammars is the removal of the selection value for products.

2.3 Mapping to UTFM

The next step is to show how classical semantics are integrated into the UTFM language and to provide the mapping from the first to the second. An example of such a translation is given in original Figure 2.1b to translation Figure 2.2:

Features. Each feature that is declared in a classical feature model is mapped to a feature type in a FMTD. These feature types are either optional $[0..1]$ or mandatory $[1..1]$.

Hierarchical Structure. The hierarchical tree structure of the classical feature models is also the basis of the UTFM language. Parent-child relationships are translated to a general group relation with a cardinality $[1..u]$. The lower bound 1 of the group cardinality will be the number of mandatory instances and the upper bound u is the sum of mandatory and optional instances.

Cross-Tree Constraints. The introduction of a scope for feature types affects the way cross-tree constraints are specified. Classical constraints were defined globally, while constraints in UTFM are written from the perspective of a constrained feature type (as

previously discussed in Section 2.2.2). Mapping classical constraints we translate *A requires B* to *A implies B*, and *A excludes B* to *(A implies not B)* and *(B implies not A)*. Propositional logic in classical feature models does not need a conversion. A feature reference is mapped by (nested) *exists* operations up to the corresponding feature type.

The configuration language of UTFM is fairly small as it specifies instances and their selection value. Selected classical features are mapped to instances of their corresponding feature type with the selection variable set to *true* and other features to instances with the values *false*, or *undecided* in case of a partial configuration.

2.4 Automated Analysis with Z3

To analyze a type declaration along with its configuration we map both in a Z3 decision problem. The *Z3 theorem prover* is written by Microsoft Research and implements a *Satisfiability Modulo Theories (SMT)* problem solver, in which different logical theories are combined and translated to a *satisfiability (SAT)* problem [16, 30]. In Z3, theories such as propositional logic, linear algebra, arrays, data structure and quantifier theory are combined and integrated with different decision procedures to form a sophisticated specification language. A Z3 decision problem (or *Z3 model*) is tested by the theorem prover for *satisfiability*. If the Z3 model is satisfiable it can be concluded that the configuration is *valid* with regard to the provided type declaration, no constraints are violated and there is at least one possible product.

A Z3 model consists of two types of expressions: *declarations* and *assertions*. Declarations define variables and functions. Asserts assign values to variables, or constrain variables by describing a relation between them. The syntax of Z3 should be read as a functional programming language, first a function name is given (for example: *=>*, *and*, *or*, *cardinality*), followed by function parameters.

Figure 2.9 shows a basic Z3 satisfiability problem, involving instances A1 and B1, and possible constraint A and B. Lines 1–2 declare the instances, Lines 4–5 assert the constraint to the model and that instance A1 is *true*. Running the problem in the prover ⁵ shows that it is satisfiable. Adding the following assertion (`(assert (not B1))`) results in an unsatisfiable problem.

```

1 (declare-const A1 Bool)
2 (declare-const B1 Bool)
3
4 (assert (and A1 B1))
5 (assert A1)
6
7 (check-sat)

```

Figure 2.9: Very basic Z3 satisfiability problem.

Variable declarations and assertions are straightforward, Z3 functions are interpreted as follows: Figure 2.10 declares function `isSelected` with a single parameter of type `Bool` that returns an `Int`. The value of the returned `Int` is decided by a *if-then-else* construct (*if x then 1 else 0; ite x 1 0*). The function returns 1 if an instance is selected, otherwise 0. The purpose of this function is explained in the next subsection.

```

1 (define-fun cardinality ((x Bool)) Int (ite x 1 0))

```

Figure 2.10: *Cardinality* function.

Z3 requires all declarations to be listed first, followed by assertions. The order within the series of declarations and asserts are provided does not matter as the model is evaluated as a whole and the state of the model does not change imperatively. After the final assert expression, the theorem prover is asked to check whether the model is satisfiable and to present a satisfying model if there is one. It is currently not possible to iterate over the satisfying models in Z3.

⁵online Z3 prover: <http://rise4fun.com/z3>

2.4.1 Translating to Z3

Functions can be nested as arguments to other functions, as in functional programming. The mapping of `GraphLibrary` to Z3 is explained in Table 2.1. The table shows the translation using snippets of the FMTD and the configuration of the running example in Figure 2.2 and Figure 2.8. The Z3 code examples that are referenced in the table use the `isSelected` function that is declared in Figure 2.10 and is explained in this subsection at item *Cardinality Evaluation*.

Instances & Instance Cardinality. For every instance in the configuration that has a selection value *true* there is a variable declared and asserted in the Z3 model. Instances that are *undecided* are only declared in the model, no value is asserted. Instances that have a *false* selection value have no translation. A *false* instance never changes the outcome of translated semantics in a Z3 model (*exists* is regarded as an *or* operation, cardinality constraints only count selected instances)⁶, and hence can be omitted in the Z3 model, however remain part of the configuration.

For every feature type, an instance cardinality variable is declared, representing the amount of selected instances. The upper- and lower bound of the cardinality are translated to assertions that constrain the bounds on the cardinality value. An example of such a mapping is demonstrated in the first row of Table 2.1 (instance: Figure 2.11 Line 1, Line 4; instance cardinality: Line 2, Lines 6–8).

Group Relationship & Group Cardinality. Each selected child instance requires their parent instance to be selected. Therefore, for every parent-child relation an implication is asserted in the model: `Child ==> Parent` (written in Z3: `(==> Child Parent)`). A group cardinality is declared in the same fashion as an instance cardinality. The second row in Table 2.1 (relation: Figure 2.12 Lines 9–10; group cardinality: Line 4, Lines 12–14) illustrates an example of this mapping.

⁶As we explain later, *forall* operations are regarded as conjunctions of implications, therefore also do not contradict this statement.

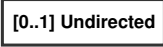
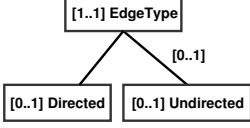
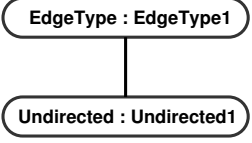
Semantics	FMTD	Configuration	Z3
Instance & Instance Cardinality			Figure 2.11
Group relation & Group Cardinality			Figure 2.12
Cross-tree Constraint	exists(Algorithm.exists(Cycle)) => exists(EdgeType.exists(Directed));		Figure 2.13

Table 2.1: Examples of mapping from UTFM to Z3.

```

1 (declare-const Undirected1 Bool)
2 (declare-const Undirected.InstanceCardinality Int)
3
4 (assert Undirected1)
5
6 (assert (= Undirected.InstanceCardinality (isSelected Undirected1)))
7 (assert (>= Undirected.InstanceCardinality 0))
8 (assert (<= Undirected.InstanceCardinality 1))

```

Figure 2.11: Instance and instance cardinality in Z3.

```

1 (declare-const EdgeType1 Bool)
2 (declare-const EdgeType1.Directed1 Bool)
3 (declare-const EdgeType1.Undirected1 Bool)
4 (declare-const EdgeType1.GroupCardinality Int)
5
6 (assert EdgeType1)
7 (assert EdgeType1.Undirected1)
8
9 (assert (=> EdgeType1.Directed1 EdgeType1))
10 (assert (=> EdgeType1.Undirected1 EdgeType1))
11
12 (assert (= EdgeType1.GroupCardinality (+ (isSelected EdgeType1.
    Directed1) (isSelected EdgeType1.Undirected1))))
13 (assert (>= EdgeType1.GroupCardinality 0))
14 (assert (<= EdgeType1.GroupCardinality 1))

```

Figure 2.12: Group and group cardinality in Z3.

Cardinality Evaluation. To evaluate the cardinality constraints of an instance or a group, the `isSelected` function is called. Defined in Figure 2.10, it has a boolean input parameter: the variable declaration of an instance (selection value), and returns an integer: 1 if the instance variable is selected (*true*) and 0 otherwise. The sum of all *isSelected* function calls on the instances involved is asserted to the corresponding cardinality variable. This cardinality variable is thereafter asserted to be equal or higher than the lower bound and equal or lower than the upper bound of the cardinality constraint specified in the type declaration. Examples on the cardinality mapping with the *isSelected* function are given in the first and second row of Table 2.1 (Instance cardinality: Figure 2.11 Line 2, Lines 6–8; group cardinality: Figure 2.12 Line 4, Lines 12–14). In the satisfying model found when validating Figure 2.12 `EdgeType1.Directed1` is *false*.

Cross-Tree Constraints. Cross-tree constraints consist of propositional formulas combined with `exists()` operations. These operations return a boolean value on whether there exists a selected instance in the configuration. An *exists* operation is translated to a logical *or* formula with a the types' declared instances as propositions. However since there are only singleton feature types, such formulas consist of a single proposition, *ie* a reference to a single instance variable. These (nested) feature type references are mapped to their unique name, through their path in the configuration. This is illustrated in row three of Table 2.1 (cross-tree constraint: Figure 2.13).

```
1 (assert (=> GraphLibrary1.Algorithm1.Cycle1 GraphLibrary1.EdgeType1.
    Directed1))
```

Figure 2.13: Cross-tree constraint in Z3.

2.4.2 Automated Analysis

With the ability to check validity of configurations with an off-the-shelf solver, we define a process for a *automated analysis operation* on UTFM models in Figure 2.14. The first step in the process, *parsing* and *validating*, requires a FMTD and a configuration. Both are parsed and type checked before they are translated to Z3, to be checked for satisfiability. After validating that there is a satisfying model, there is the choice to perform *constraint propagation* with or without *unfolding* the configuration. The propagation process returns a new configuration (Configuration*). The configuration could be configured further and iterate through the analysis process. The three steps for *automated analysis* are: *validating*, *unfolding*, and *constraint propagation*.

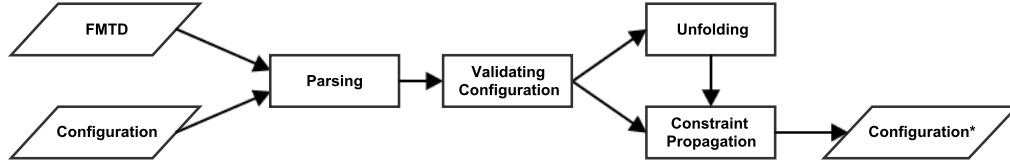


Figure 2.14: Automated analysis process.

Parsing & Validating. After parsing and type checking the type declaration and configuration are interpreted. Using the mapping rules discussed in the previous subsection the model is translated to Z3. If the configuration is *invalid* with regard to the provided type declaration the analysis process is terminated. Otherwise the process continues with *constraint propagation* or the *unfolding algorithm*. As the latter is followed by *constraint propagation* as well, we elaborate on this operation first.

Constraint Propagation. A fundamental concept in classical feature modeling tools is for users to never specify an configuration that fails to satisfy all feature model constraints [18]. Only valid partial configurations can be specified. This capability is accomplished by *constraint propagation*: all facts that are inferable from a given set of facts are determined (so that a user cannot specify facts that contradict with feature model constraints). Constraint

propagation is performed using Janota’s algorithm [21]. The referenced algorithm reasons on features with a selection value, in UTFM the analysis is performed on instances, this does not affect the algorithm. The Janota’s algorithm is summarized in the following points:

- if there is a valid configuration for which an *undecided* instance can be *true* and another valid configuration for which that instance can be *false*: the instance remains *undecided*. That is, a user has the freedom to decide whether or not to select the instance at a future time.
- if the only remaining valid configurations require the instance to be *true*, the instance is asserted *true*.
- if the only remaining valid configurations require the instance to be *false*, the instance is asserted *false*.
- there are no valid configurations (no matter the selection value of the instance), then there is a contradiction in the feature model or with the current configuration as no products are possible.

The UTFM interpretation of the algorithm asserts for each *undecided* instance in the configuration consecutively *true* and *false* and validates the models in the Z3 prover. Based on returned satisfiability of the models the instance becomes configured or remains unconfigured.

In Figure 2.15 we provide a configuration of the `GraphLibrary` FMTD (Figure 2.2). In the configuration there are several *undecided* instances which the algorithm will try to configure through *constraint propagation*. Figure 2.16 shows the result from the algorithm, three instances are propagated, two remain *undecided*. The first cross-tree constraint implies that the instances `Undirected1` and `Weighted1` must be propagated to *true*. Due to this propagation, `Directed1` is propagated to *false*. Because the group cardinality of `EdgeType1` does not permit `Directed1` to be *true* as well. As a result the instance `Cycle1` is also

propagated to *false*, this instance is not allowed to be *true*, due to the second cross-tree constraint. It implies that if *Cycle1* is *true* there has to be a *Directed* instance selected, which is not possible in the current configuration. Both *Prim1* and *Kruskal1* remain *undecided* as *valid* configurations are possible with either instance selected while the other is not selected.

```

1  GraphLibrary : GraphLibrary1
2    group:
3      EdgeType : EdgeType1
4        group: {
5          (undecided) Undirected : Undirected1
6          (undecided) Directed : Directed1
7        }
8      (undecided) Weighted : Weighted1
9      Algorithm : Algorithm1
10     group:
11       (undecided) Cycle : Cycle1
12       MST : MST1
13       group:
14         (undecided) Prim : Prim1
15         (undecided) Kruskal : Kruskal1

```

Figure 2.15: Configuration of Figure 2.7.

```

1  GraphLibrary : GraphLibrary1
2    group:
3      EdgeType : EdgeType1
4        group: {
5          Undirected : Undirected1
6          (false) Directed : Directed1
7        }
8      Weighted : Weighted1
9      Algorithm : Algorithm1
10     group:
11       (false) Cycle : Cycle1
12       MST : MST1
13       group:
14         (undecided) Prim : Prim1
15         (undecided) Kruskal : Kruskal1

```

Figure 2.16: Propagated configuration of Figure 2.15.

Unfolding. For the constraint propagation process to do anything there need to be unconfigured instances (explicitly declared *undecided* instances). Additionally, in order for the

process to considers all possible propagations, all possible instances have to be declared in a configuration. To obtain such a configuration we introduce the *unfolding* process, to generate maximum instances for all feature types. The algorithm generates instances up to the upper bound of a feature types' instance cardinality. We assume that this is the maximum, even though in theory there are implicitly, infinitely many *undecided* instances. The process is called *unfolding* as the instance tree will be unfolded in every node.

Algorithm 2.1 Unfolding Algorithm

```

1: procedure UNFOLDING(Instance  $i$ )
2:   FeatureType  $ft \leftarrow \text{getFeatureType}(i)$ 
3:   FeatureType[ ]  $group \leftarrow \text{getChildTypes}(ft)$ 
4:   for each  $childType$  in  $group$  do
5:     (Int  $lower$ , Int  $upper$ )  $\leftarrow \text{getInstanceCardinality}(childType)$ 
6:     Instance[ ]  $instances \leftarrow \text{getChildInstances}(i, childType)$ 
7:     for  $count(instances) < upper$  do
8:       Instance  $new \leftarrow \text{generateInstance}(childType, \text{"undecided"})$ 
9:        $instances \leftarrow instances \cup [new]$ 
10:     $\text{setChildInstances}(i, childType, instances)$ 
11:    for each  $instance$  in  $instances$  do
12:      if  $\text{selection}(instance) = \text{"true"}$  or  $\text{selection}(instance) = \text{"undecided"}$  then
13:        UNFOLDING( $instance$  of feature tree)
14: main
15: Instance  $i \leftarrow \text{root instance}$ 
16: UNFOLDING( $i$ )

```

The process is based on three steps, and starts in the root instance of a configuration:

- (1) For all child feature types in the group relation of the instance, the algorithm checks if there are as many instances configured as the upper bound of the instance cardinality as the feature types permits;
- (2) If this is not the case, instances of the child feature type are generated with the selection value set to *undecided*. For feature type A with instance cardinality $[1..u]$, instances are generated until A_u . In classical feature models there is just A_1 .

(3) The process recursively proceeds to step 1 for all *true* or *undecided* instances⁷

The described unfolding process is provided in pseudo code in Algorithm 2.1. The algorithm uses the following functions:

getFeatureType(Instance) Returns the feature type of the provided instance;

getChildTypes(FeatureType) Returns the set of feature types that are children of the given feature type;

getInstanceCardinality(FeatureType) Returns the instance cardinality of a feature type;

getChildInstances(Instance, FeatureType) Returns the instances of a particular feature type from the group relation of the provided instance, this is a set of instances;

setChildInstances(Instance, FeatureType, Instances) Assigns the set of instances for FeatureType of the parent Instance to Instances;

generateInstance(FeatureType, Selection) Returns a generated instance of a feature type, the instance has a generated name, and the given selection value;

selection(Instance) Returns the selection value of an instance.

The constraint language uses *exists*, which suggests that we need to introduce quantifications to the problem model. By unfolding and mapping *exists*(A) predicates to Z3 disjunctive assertions $A_1 \vee A_2 \vee \dots \vee A_u$, we avoid first-order logic in Z3 decision problems and use equivalent propositional formulae. Using the *unfolding* algorithm during automated analysis of configuration, all possible products are considered against the set of specified constraints in the type declaration. Our proposed semantics on instances have been reduced to a well-known (and simpler) problem that makes use of propositional logic [4].

The configuration provided in Figure 2.16 is unfolded in Figure 2.17. Notice how the generated instance have a randomized name. By configuring all *undecided* instances to *false*

⁷*False* instances do not have to be unfolded. Possible child instances are all propagated to false and will never be selected for the product.

the configuration is completely configured. We can regard this as the end of a configuration process. By using this configuration as input for the unfolding and propagating operations an isomorphic configuration is returned (Figure 2.18), no instances are generated, none are propagated. By omitting everything but the selected instances in the configuration we obtain the product specification (*full* configuration) illustrated in Figure 2.19 (notice the similarity with Figure 2.7, however this configuration represents a set of products).

```

1  GraphLibrary : GraphLibrary1
2    group:
3      EdgeType : EdgeType1
4        group: {
5          Undirected : Undirected1
6          (false) Directed : Directed1
7        }
8      (undecided) Search : Search54356
9        group: {
10         (undecided) BFS : BFS89334
11         (undecided) DFS : DFS26730
12       }
13    Weighted : Weighted1
14    Algorithm : Algorithm1
15      group:
16        (false) Cycle : Cycle1
17        (undecided) ShortestPath : ShortestPath92124
18        MST : MST1
19          group: {
20            (undecided) Prim : Prim1
21            (undecided) Kruskal : Kruskal1
22          }
23        (undecided) Transpose : Transpose42166

```

Figure 2.17: Unfolded configuration of Figure 2.16.

```

1  GraphLibrary : GraphLibrary1
2    group:
3      EdgeType : EdgeType1
4      group: {
5        Undirected : Undirected1
6        (false) Directed : Directed1
7      }
8      (false) Search : Search54356
9      Weighted : Weighted1
10     Algorithm : Algorithm1
11     group:
12       (false) Cycle : Cycle1
13       (false) ShortestPath : ShortestPath92124
14       MST : MST1
15       group: {
16         (true) Prim : Prim1
17         (false) Kruskal : Kruskal1
18       }
19       (false) Transpose : Transpose42166

```

Figure 2.18: Full configuration of Figure 2.17.

```

1  GraphLibrary : GraphLibrary1
2    group:
3      EdgeType : EdgeType1
4      group: {
5        Undirected : Undirected1
6      }
7      Weighted : Weighted1
8      Algorithm : Algorithm1
9      group:
10       MST : MST1
11       group:
12       Prim : Prim1

```

Figure 2.19: Product specification of Figure 2.18.

Chapter 3

Feature Replication

We allow feature replication to UTFM and describe how it affects our type declarations and the analysis of its configurations:

1. We explain how previous researchers have defined feature replication;
2. We give our interpretation of replication and how is it manifested in UTFM;
3. We show the effect of replication on the analysis of models and their translation to Z3.

Feature replication has a significant influence on the design of UTFM, which explains design decisions made in the previous chapter as much as it forces restrictions on the language, that will surface when we extend UTFM with feature attributes in Chapter 4.

3.1 Replication of Features

Czarnecki et al. in 2005 [14] first introduced the notion of having multiple clones of a feature. Feature cardinality allowed the *cloning* of classical features, in UTFM vocabulary this is translated to: *feature replication* which allows there to be *multiple instances* of a feature type in a configuration. Czarnecki's proposal distinguishes differences between features, not all features are clonable. There are features in group relations and solitary features,

where only the solitary features are clonable. Examples of constraints on features with replication were provided and written in *Object Constraint Language (OCL)*, however no language for cross-tree constraints (other than examples in OCL) was provided. Also no complete mapping of semantics and constraints to OCL was provided. Figure 3.1 illustrates a proposed example of feature replication; we overlook the presence of feature attributes and references to feature models. In this example, it is possible to replicate, for example, the feature `StoreFront` or `PaymentMethod`, and for any of their replicas (or instances) there can be multiple grandchildren. For each of the `StoreFront` replica's `PaymentMethodRef` and `ShippingMethodRef` can be replicated. This is regarded as nested replication.

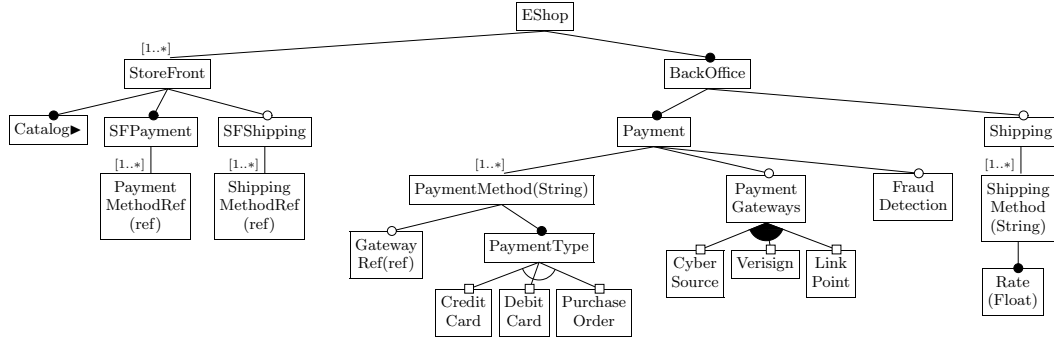


Figure 3.1: Feature model with replication from [15].

The *Common Variability Language (CVL)* [25] was recently proposed and abandoned as an OMG standard for feature modeling. CVL entails feature replication, feature attributes, along with a constraint language for cross-tree constraints. Having only an informal description, it was not specified how nested replicated features were to be evaluated with regard to model constraints. Only a few examples were given; a formal specification of CVL was not provided.

3.2 Multiple Instances in UTFM

The structure of UTFM, explained in the previous chapter does not have to be changed to support multiple instances of features; just lifting the restriction of allowing only singleton features is needed. The foundation of UTFM to support replication was laid out in the previous chapter: (1) *instance cardinality*; (2) the introduction of *operations* on feature types in the constraint language; (3) and a *local scope* for feature types and constraints rather than global cross-tree constraints.

3.2.1 Type Declaration

As described in Chapter 2, cardinalities are evaluated against the number of selected instances in a configuration in the scope of a parent instance. This is however an explicit choice as nested feature replication can be interpreted in different ways. The paper by Michel et al. published in 2011 [26], describes different interpretations of the semantics of group and instance cardinalities with regard to feature replication. They provide a two by two matrix with on the axis:

Level. The level is set to either *global* or *local*. The cardinality of a feature model is counted *globally* for the entire model, or it is counted *locally* for each instance.

Scope. The scope is set to either *types* or *instances*. Does the group cardinality is applied on *instances* or *types*.

Figure 3.2 shows on the right different interpretations of the type declaration provided on the left, based on *level* and *scope*. The classification is provided in [26] and is translated to UTFM for consistency. The interpretations are valid configurations that are possible under the assumed semantics. In the publication by Michel et al. they choose to interpret cardinality with the *level* set to *local* and the *scope* set to *types* (bottom right in Figure 3.2). Our interpretation also has the *level* set to *local* but differ as we apply cardinality to *instances*

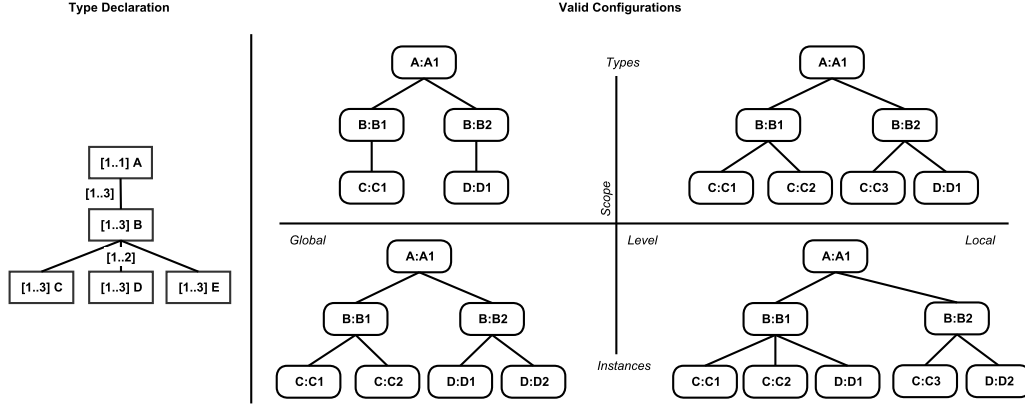


Figure 3.2: Classification of different cardinality interpretations.

(top right). The motivation for their choice was based on being able to specify optionality for feature types. They argue that this is closer to the classical interpretation of optional feature. Our motivation to set the *scope* to *instances* is that the language is more expressible. The optionality of feature types is easily translated with the introduction of extra optional singleton feature types. While the other way around, restricting the total amount of child instances with a *scope* set to *types* can't be expressed easily (the same feature types need to be declared multiple times, or complex constraints need to be introduced to the type declaration).

In the publications by Czarnecki et al. [14, 15] on cardinalities the interpretation of nested replication is not thoroughly explained, this is also omitted in the CVL proposal [25]. Therefore it is impossible for us to demonstrate that we preserved prior semantics for feature replication in UTFM. Or make assumptions on their interpretation of *level* and *scope*.

3.2.2 Constraint Language

Given that multiple instances of a feature type is allowed, the logical operation `forall` can now be introduced to the UTFM constraint language:

Forall(FeatureType). Operation *forall* is a *boolean* expression and returns *true* if an ex-

pression is evaluated *true* for every *selected* instance of a feature type that is referenced, otherwise *false*. `FeatureType` can be any child feature type of the referenced parent type with either a nested operation or an expression. In the case there are no selected instances *forall* returns *true*, as no constraint is violated.¹

An *exists* operation is the logical equivalent of a disjunction between configured instances.² An *forall* operation is equivalent to a conjunction of implications for configured instances. These implications are interpreted as: if the instance is selected in a configuration then the expression given in the *forall* should hold. A *forall* operation constrains all *true* selected instances of a feature type.

Let P, C, G be feature types, where P is a parent, C its child, and G a child of C (equivalently G is a grandchild of P). Consider the expression `forall(C)`. It means: for a given P instance, all selected C instances should be *true* (“ C_i implies C_i ”). Such constraints are trivial as they are always *true*. *Forall* with nested operations or expressions such as `forall(C.exists(G))`, are interpreted as: for a given P instance, all of its selected C instances must have a selected G instance (“ C_i implies $C_i.exists(G)$ ”). Figure 3.3 (graphical) and Figure 3.4 (textual) illustrate more complex constraints using *forall* operations. The constraints stated in the examples are described in natural language as follows:

- (a) For a given instance of P , if an instance of X is selected then for every *selected* child C a grandchild G is required to be *selected*.
- (b) For a given instance of P , either all *selected* C child instances have a X instance configured to *true* or there is an Y instance *selected*.

The type declaration of Figure 3.3a is provided in Figure 3.4a, and for Figure 3.3b in Figure 3.4b.

¹These semantics can be compared to how universal quantifiers in first-order logic in the beginning are always valued as *true* and therefore are evaluated as *true* when quantified over an empty domain [29].

²Remember that not all instances in a configuration need to be selected.

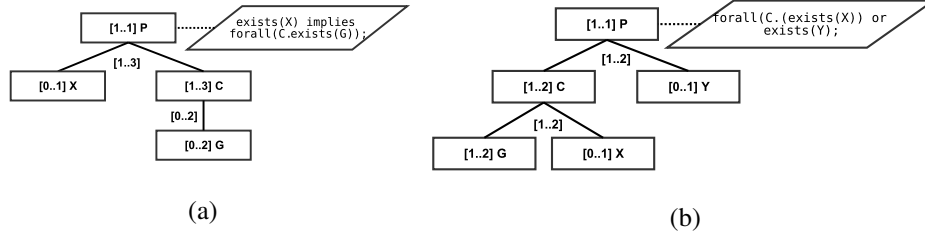


Figure 3.3: *Forall* operation examples.

<pre> 1 [1..1] P 2 constraints: 3 exists(X) implies 4 forall(C.exists(G)) 5); 6 group [1..3]: 7 [0..1] X 8 [1..3] C 9 group [0..2]: 10 [0..2] G </pre> (a)	<pre> 1 [1..1] P 2 constraints: 3 forall(C.exists(X)) or 4 exists(Y); 5 group [1..3]: 6 [1..3] C 7 group [1..2]: { 8 [1..2] G 9 [0..1] X 10 } 11 [0..1] Y </pre> (b)
---	--

Figure 3.4: Type declarations for Figure 3.3.

3.2.3 Configuration

No new syntax needs to be introduced to FMTD to specify multiple instances. Where ever singular instances were declared, multiple instances can be declared and for each of their child instances, recursively. The new constraints with regard to multiple instances are explained gradually through configuration examples of Figure 3.3b provided in Figure 3.5.

The first of the configurations, Figure 3.5a, is a *valid partial* configuration, as with further configuration of instances one or more products can be described. Implicitly there are *undecided* instances of the X and Y feature types for which it is still to configure them to *true*. Therefore the constraint `forall(C.exists(X)) or exists(Y);` is not violated, rendering the configuration otherwise to *invalid*. Another *valid* configuration is provided in Figure 3.5b. The difference between Figure 3.5a and Figure 3.5b is that the second can be interpreted as a

<pre> 1 P:P1 2 group: 3 C:C1 4 group: 5 G:G1 </pre> (a)	<pre> 1 P:P1 2 group: 3 C:C1 4 group: { 5 G:G1 6 G:G2 7 } 8 Y:Y1 </pre> (b)
<pre> 1 P:P1 2 group: 3 C:C1 4 group: { 5 G:G1 6 G:G2 7 } 8 C:C2 9 group: { 10 G:G3 11 G:G4 12 } 13 C:C3 14 group: { 15 G:G5 16 G:G6 17 } </pre> (c)	<pre> 1 P:P1 2 group: 3 C:C1 4 group: { 5 G:G1 6 (false) G:G2 7 X:X1 8 } 9 (undecided) C:C2 10 group: { 11 (undecided) X:X2 12 } 13 (undecided) Y:Y1 </pre> (d)

Figure 3.5: Multiple configurations of Figure 3.3b.

full configuration, as a product, while the first can not³.

Figure 3.5c provides a configuration that is *invalid*. The constraint of P1 is violated as no instance of X or Y has been selected in configuration, while the instance cardinalities do not allow the selecting of more instances to satisfy the constraint. Finally, Figure 3.5d also shows a *valid partial* configuration, however, besides *selected* instances, we removed the instance G2 (configured to *false*), and explicitly declared the *undecided* instances C2, X2 and Y1.

³Remember that for product specifications only *selected* instances are stated, and implicitly all other instances are assumed *not* to be *selected*.

3.3 Automated Analysis with Z3

The process of automated analysis of configurations remains as stated in the previous chapter, Section 2.4.1. We translate the *forall* operation and clarify constraint propagation through an example.

3.3.1 Translating to Z3

We show in Chapter 2 how *exists* operations are defined and translated to Z3 for singular instances. The translation of an *exists*(C) operation with four C instances is provided in Figure 3.6; an propositional *or* formula with selection variables as primitive terms. The *forall* operation translates in a similar fashion to a conjunction of implications. The implication consists of the instances of the feature type to ensure that they are selected, and if so, the consequent (the second proposition) must be true. This proposition is either a nested boolean operation or an expression (just a feature type reference results in a trivial implication; X_i implies X_i).

```
1  (assert
2    (or P1.C1
3        (or P1.C2
4            (or P1.C3 P1.C4)
5        ) ) )
```

Figure 3.6: Z3 translation of an *exists* operation on a non singular feature type.

The translation of the constraint of Figure 3.3b, *forall*(C.*exists*(X)) or *exists*(Y);, that uses a *forall* operation, is provided in Figure 3.7. The translation is based on the configuration specified in Figure 3.5d, in which two C instances are declared, with each an X instance. Translated to Z3 the assert on the Z3 model representing the *forall* operation, consists of a conjunction of two implications. These conjunctions are, along with the Y instance, propositions in an *or* expression. For the stated assert statement to be *true* the Z3 theorem prover will assign either C2 and X2 to *true*, Y1 to *true* or all three instances to *false* (in the configuration C1 and X1 are already *selected*, other instances are *undecided*).

```

1  (assert
2    (or
3      (and
4        (=> P1.C1 P1.C1.X1)
5        (and
6          (=> P1.C2 P1.C2.X2)))
7      P1.Y1
8    )
9  )

```

Figure 3.7: Z3 translation of cross-tree constraint of Figure 3.3b using Figure 3.5d .

3.3.2 Constraint Propagation

We continue using the type declaration of Figure 3.3b as a running example, for now to illustrate the unfolding and propagating process. The unfolded configuration of Figure 3.5a, is provided in Figure 3.8a. Notice that the generated *undecided* instance have a randomized instance name. Also note that the unfolding algorithm only generates instances up to the maximum of the instance cardinality while in theory it could generate infinitely *undecided* instances, in the process of making the undecided explicit.

In this unfolded configuration we select the instances G93423, C64210 and X81274 and run the propagating process, the result is provided in Figure 3.8b. Due to the constraint that is declared in the type declaration the Y instance is propagated to *true* (Line 21). Other instances are propagated to *false* (Line 7, Lines 15–19), the group cardinalities do not permit more instances to be selected. The instances G01182 and G28399 remain *undecided* as the algorithm is not able to determine which of these is certainly selected for the configuration.


```

1 P:P1
2   group:
3     C:C1
4       group: {
5         G:G1
6         (undecided) G:G93423
7         (undecided) X:X54744
8       }
9     (undecided) C:C64210
10    group: {
11      (undecided) G:G01182
12      (undecided) G:G28399
13      (undecided) X:X81274
14    }
15    (undecided) C:C91828
16    group: {
17      (undecided) G:G56252
18      (undecided) G:G11563
19      (undecided) X:X79162
20    }
21    (undecided) Y:Y59676

```

(a) Unfolded configuration of Figure 3.5a.

```

1 P:P1
2   group:
3     C:C1
4       group: {
5         G:G1
6         G:G93423
7         (false) X:X54744
8       }
9     C:C64210
10    group: {
11      (undecided) G:G01182
12      (undecided) G:G28399
13      X:X81274
14    }
15    (false) C:C91828
16    group: {
17      (false) G:G56252
18      (false) G:G11563
19      (false) X:X79162
20    }
21    Y:Y59676

```

(b) Propagated configuration.

Figure 3.8

Chapter 4

Attributed Feature Models

We extend the UTFM language with attributes in this chapter. We describe:

1. The classical semantics for attributed feature models;
2. How attributes extend the UTFM language;
3. A translation of classical attributed feature models to UTFM;
4. And how attribute semantics of UTFM translate to Z3 and work in automated analysis of attributed feature models.

In these section we elaborate on feature attributes in UTFM, an important building block for the constraint language of UTFM.

4.1 Classical Attributed Feature Models

As with classical feature models in Chapter 2, we require UTFM to be backwards compatible with *Attributed Feature Models* published previously. Classical feature models did not allow features to have attributes, even though their presence and usage had been discussed by Kang et al. in 1998 as non-functional features [24]. *Attributed* or *advanced* feature models

were first introduced by Czarnecki et al. [14, 15] and Benavides et al. in 2005 [4]. They added *arithmetic* and *string* attributes and constraints on these attributes to classical notion of feature modeling.

Arithmetic & String Attributes. From these proposals, each with differences in syntax and semantics, we summarize that each attribute has a name, domain and value and is related to a single feature. The domains for arithmetic attributes are discrete or continuous: integer or real. In [14], string attributes are introduced to specify the names of options used in the implementation of products. An example of Benavides's attributed feature model is shown in Figure 4.1.

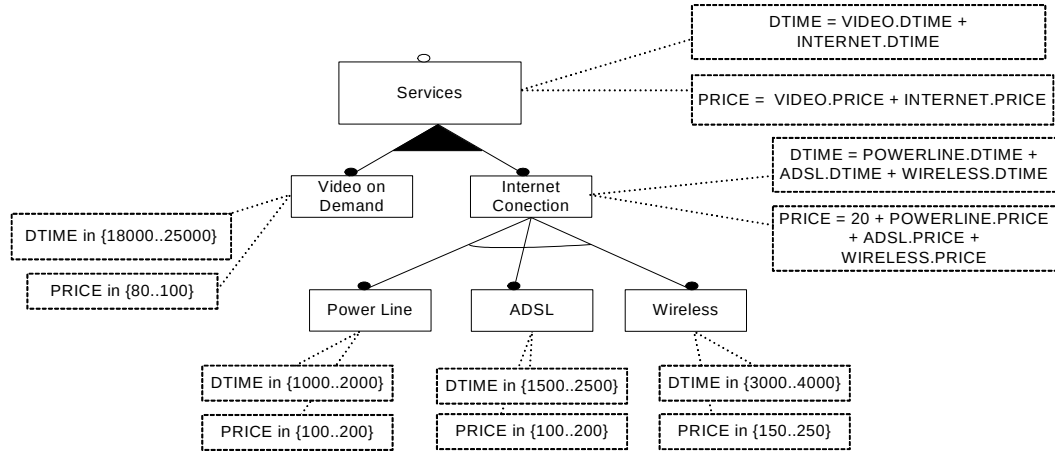


Figure 4.1: Attributed feature model by Benavides et al.[7].

Arithmetic constraints. As Figure 4.1 shows, declarations of an attributes can be constrained by a range of possible values. The constrained specifies an upper- and a lower bound of the valid values of the attribute. Furthermore, attribute values can be constrained to a certain value that is based on an arithmetic expression defined using arithmetic attributes of other features. Besides the introduction of string attributes there are no string constraints provided in [15].

4.2 Attributes in UTFM

Introducing attributes into the UTFM language requires an extension of the syntax and semantics previously defined. Attributes are declared in the FMTD and are assigned values during configuration. For each instance, of a feature type, the attribute set (declared and constrained in the type declaration) has values. This is consistent with prior work on attributes in classical feature models where features just had singular instances.

4.2.1 Type Declaration Extensions for Attributes

Attribute declarations are specified in the type declaration and consist of a *name* and a *type* (domain of the attribute). Four attribute types are distinguished in UTFM: *integer*, *real*, *boolean* and *string*. All attributes (regardless of their type) must have unique names with respect to their feature type. When an instance is created all attributes have their value set to *undecided*, unless a specific value is provided in the configuration. An *undecided* value can be determined by constraint propagation or configured at a later stage.¹ Each attribute can be referenced in arithmetic constraints of the FMTD.

Integer Attributes. *Integer attributes* are declared as type `Int`. In configurations of a type declaration integer values are assigned to these attributes.

Real Attributes. *Real attributes* (declared as `Real`) are used in the same fashion as *integer* attributes. In arithmetic expressions both *integer* and *real* attributes can be mixed; type casting an *integer* to a *real* in a configuration does not change the value, the other way around will result in dropping the decimal value of the *real*, the value is not rounded to the nearest *integer*.

Boolean Attributes. Boolean attributes (declared as `Bool`) have not thoroughly made their way into attributed feature models. Yet boolean attributes play an important role in

¹Z3 allows undecided attributes (integer, real and boolean) as the prover will look for possible values for the attributes through resolution to get to a satisfiable model.

substitution possibilities in the FMTD constraint language, which is explained in the *Substitution* item of Section 4.2.2. Furthermore boolean attributes can be used to reduce² feature models in a process explained by Classen et al. [11], in which features types are replaced by boolean attributes.

String Attributes. String attributes (declared as *String*) provide possibilities to specify options, names, descriptions, etc. for instances, providing extra information for the implementation of a product. These attributes can't be referenced in constraints.

A portion of Figure 4.1 (namely the *InternetConnection* subtree) is shown in Figure 4.2, where attribute declarations are specified in the same parallelogram as cross-tree constraints, separated by a horizontal line; attributes above the line, constraints thereunder. The UTFM syntax to declare attributes is an “*attributes:*” statement followed by attribute declarations. The translation of Figure 4.2 to UTFM is given in Figure 4.3. In the FMTD a *string* and *boolean* attribute have been added to *InternetConnection* to illustrate the possibilities.

In anticipation of the next subsection that elaborates on cross-tree constraints and UTFM constraint language, we briefly note the following on the constraints in the Figure 4.2. It is impossible to directly reference the values of attributes of instances in a type declaration, therefore a *sum* operation is called on a feature type. *Sum* returns the sum of all *price* attributes of the *selected* instances of the referenced feature type (either: *Powerline*, *ADSL* or *Wireless*).

4.2.2 Constraint Language Extensions for Attributes

The FMTD constraint language is extended with new operations: *assign*, *equality* and *inequality expressions*.

Assign Expression. The value of an attribute can be based on an expression. As in other languages, the syntax for assigning a value is: *name* = *expression*;;, where *name* is

²*Reducing* a feature model is reducing the total number of feature types in a model and the (maximum or average) depth of the model.

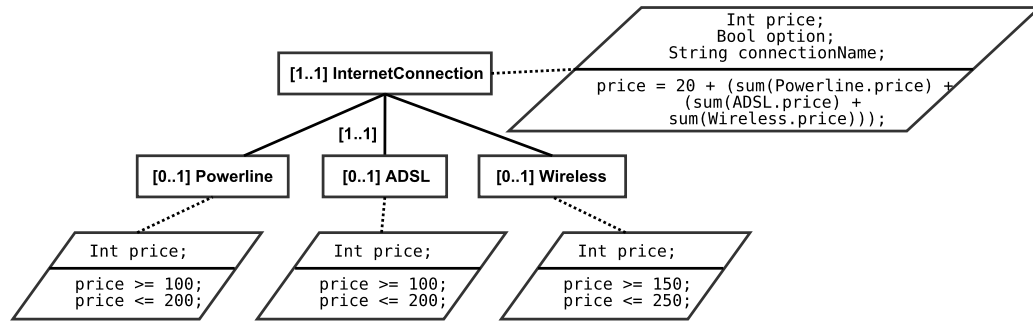


Figure 4.2: Part of the graphical FTMD representation of Service feature model (Figure 4.1).

```

1  [1..1] InternetConnection
2    attributes:
3      Int price;
4      Bool option;
5      String connectionName;
6    constraints:
7      price = 20 + (sum(Powerline.price) + (sum(ADSL.price) + sum(
8        Wireless.price)));
9    group [1..1]:
10     [0..1] Powerline
11       attributes:
12         Int price;
13       constraints:
14         price >= 100;
15         price <= 200;
16     [0..1] ADSL
17       attributes:
18         Int price;
19       constraints:
20         price >= 100;
21         price <= 200;
22     [0..1] Wireless
23       attributes:
24         Int price;
25       constraints:
26         price >= 150;
27         price <= 250;

```

Figure 4.3: Part of the FMTD of Service feature model (Figure 4.1).

the name of an attribute of the feature type and *expression* is an expression that returns a value that is in the domain of the attribute. An assign expression is considered

a constraint that is required to be *true* for a valid configuration. Key to an assign expression, in contrast to an equality expression, is that it deduces a value for an attribute. Assign expressions are not allowed inside other expressions. Figure 4.3 Line 7 illustrates a assign expression, other more basic assignments are:

```
price = 205;

connectionName = "TestConnection";

options = false;
```

Equality Expression. The syntax of an equality expression is `expression == expression`.

The difference is that on the left side of the equality sign there is an expression that is evaluated against the right side. Both left and right expressions are required to be of the same type and the equality expression itself returns a *boolean* value. Equality expressions are allowed as constraints on their own or as propositions in other expressions. An example of an equality expression is:

```
(sum(Video.price) + sum(Internet.price)) == 300;
```

Inequality Expression. Inequality expressions compare two expressions and return a *boolean* value. Possible arithmetic inequality operations are: *less than* (`<`), *Less than or equal* (`<=`), *greater than* (`>`), or *greater than or equal* (`>=`), and for both arithmetic and boolean expressions *not equal to* (`!=`). These expressions are used in the same way as equality expressions. Examples of inequality are Figure 4.3 Lines 13–14, Lines 19–20, Lines 25–26.

For arithmetic attributes and constraints, we introduce *arithmetic operations*. UTFM implements the following: *sum*, *max*, and *min*, as well as new semantics for the *boolean operations*:

exists and *forall*. The context free grammar provided in Figure 2.3 has been extended with the attribute constraints in Figure 4.4, the complete grammar for the constraint language is provided in Appendix A.4.

sum(FeatureType.Attribute). The operation `sum` has an attribute reference as parameter and returns an *integer*. Given parent feature type P , and child C with attribute a , the `sum` operation written in a constraint of P would be: `sum(C.a)`. *Sum* returns the sum of all a attribute values of C_j instances that are selected in a configuration.³ In case there are no selected instances of C , `sum` returns 0;

max(FeatureType.Attribute). The operation `max(C.a)` returns the maximum value of the a attributes of the selected C instances. In the case that there are no selected C instances, `max` returns 0;

max(ArithmeticExpr, ArithmeticExpr). It is also possible to call a `max` operation with two *arithmetic* expressions as parameters. Such expressions can either be an `Int` or `Real` attribute or value, or more complex *arithmetic* expression. The largest value of either of the expressions is returned;

min(FeatureType.Attribute). The `min` operation is similar to the `max` operation in any way, except that it returns the lowest value of the attributes that are referenced;

min(ArithmeticExpr, ArithmeticExpr) Also the `min` operation with *arithmetic* expressions as parameters is similar to `max`, however it returns the lowest evaluation of both expressions.

exists(FeatureType.Attribute). Besides the introduction of *arithmetic* operation the *boolean* operation `exists` is extended with additional semantics too. If child C has an `Bool` attribute b , then an operation `exists(C.b)` can be specified,, or there are

³Instances of C that are *undecided* influence the return value of `sum(C.a)`. Value optimization of arithmetic attributes is not possible yet, in UTFM as well in the current stable version of the Z3 theorem prover (4.3.0). We mention optimization as future work in Section 8.2.2.

no *selected* C instances, that returns *true* if there exists a *selected* C instance that has a b attribute that is *true*. If no such an instance of C exists, or there are no *selected* C instances, the operation returns *false*.

forall(FeatureType.Attribute). As for exists, forall(C.b) returns *true* if for every selected C instance attribute b is *true*, otherwise *false*. If there are no *selected* C instances, forall returns *true*.

To elaborate more on the possibilities of *sum* operations we introduce parent feature P, child C and D both with an attribute x and grandchild G (child of C) with attribute y. First we write a constraint in natural language followed by its equivalent in UTM.

- The sum of all x attributes of every selected C instance of a P instance is required to be less than 100:

$$\text{sum}(C.x) < 100;$$

- For every instance of P, the sum of the x values of its children plus the sum of the y values of its G grandchildren is greater than 200:

$$(\text{sum}(C.x) + \text{sum}(C.\text{sum}(G.y))) > 200;$$

- For every P instance, the maximum of the x attributes of C and D can not exceed 100:

$$(\max(\max(C.x), \max(D.x))) \leq 100;$$

Resemblance with Attribute Grammar

Chapter 2 introduced a *local scope* to feature types, with regard to how constraints and attributes are specified. This decision, to introduce a limited scope, has a big impact on writing constraints. A local scope, in contrast with a global scope for constraints, is necessary

```

1  <Constraint> ::= ... | <String> "=" <BooleanExpr> ";"
2    | <String> "=" <BooleanProp> ";"
3    | <String> "=" <ArithExpr> ";"
4    | <String> "=" <ArithProp> ";"
5  <BooleanExpr> ::= ... | <BooleanProp>
6    | <BooleanProp> "=" <BooleanProp>
7    | <BooleanProp> "!=" <BooleanProp>
8    | <ArithProp> "==" <ArithProp>
9    | <ArithProp> "!=" <ArithProp>
10   | <ArithProp> "<=" <ArithProp>
11   | <ArithProp> "<" <ArithProp>
12   | <ArithProp> ">=" <ArithProp>
13   | <ArithProp> ">" <ArithProp>
14  <BooleanProp> ::= <BooleanOp>
15    | <AttributeRef>
16    | <ParentRef>
17    | <Bool>
18    | "(" <BooleanExpr> ")"
19  <BooleanOp> ::= ... | "exists(" <FeatureRef> "." <AttributeRef> ")"
20    | "exists(" <FeatureRef> "." <BooleanExpr> ")"
21    | "forall(" <FeatureRef> "." <AttributeRef> ")"
22    | "forall(" <FeatureRef> "." <BooleanExpr> ")"
23  <ArithExpr> ::= <ArithProp> "+" <ArithProp>
24    | <ArithProp> "-" <ArithProp>
25    | <ArithProp> "*" <ArithProp>
26    | <ArithProp> "/" <ArithProp>
27  <ArithProp> ::= <ArithOp>
28    | <AttributeRef>
29    | <ParentRef>
30    | <Number>
31    | "(" <ArithExpr> ")"
32  <ArithOp> ::= "sum(" <FeatureRef> "." <AttributeRef> ")"
33    | "sum(" <FeatureRef> "." <ArithOp> ")"
34    | "sum(" <FeatureRef> "." <ArithExpr> ")"
35    | "max(" <FeatureRef> "." <AttributeRef> ")"
36    | "max(" <FeatureRef> "." <ArithOp> ")"
37    | "max(" <FeatureRef> "." <ArithExpr> ")"
38    | "max(" <ArithProp> "," <ArithProp> ")"
39    | "min(" <FeatureRef> "." <AttributeRef> ")"
40    | "min(" <FeatureRef> "." <ArithOp> ")"
41    | "min(" <FeatureRef> "." <ArithExpr> ")"
42    | "min(" <ArithProp> "," <ArithProp> ")"
43  <ParentRef> ::= "parent." <AttributeRef>
44    | "parent." <ParentRef>
45  <AttributeRef> ::= <String>
46  <Number> ::= <Integer> | <Real>

```

Figure 4.4: Constraint language grammar of Figure 2.3 extended to support attributes.

as it enforces to describe paths, the context of references, in constraints. Mainly due nested replication introduces the same feature types in different branches a context is necessary. Also our adopted name convention, only requiring unique names with regard to the parent relation, requires explicit specified context for constraints, considering global references are not unique.

Constraints are declared in the scope of a feature type where they are only allowed to reference their own attributes, their children attributes, and their parents attributes. A reference to a parent attribute is made by stating *parent* followed by the attribute name of the parent feature type, for instance: `parent.attribute`. By allowing references only to a parent or child attributes for each feature type, the feature model has similarities with an *attribute grammar*, in which there are *synthesized*- and *inherited attributes*. In attribute grammars, attributes are defined for grammar rules. These attributes are allowed to use the synthesized values from their non-terminal symbols, or the inherited values given by their parent node. UTFM attributes can be interpreted as, while having an attribute value, having a synthesizable and inheritable value. However we allow both to be used intertwined, however being less restrictive as formal attribute grammars. We take notice feature models in UTFM resemble attribute grammars, which is in line with previous publications that showed classical feature models can be written as grammars [4]. We conclude that feature models in UTFM can be regarded as attributed grammars.

Substitution

In contrast to what we just explained, we've seen in previous examples, that it is possible to nest operations and expressions in constraints. This seems contradicting with the informal attributed grammar semantics. The nesting of operations and expressions is allowed by the rules of *substitution*. It is allowed to *substitute* attributes in a reference, with the expression that assigns a value to the attribute, if defined as an assign expression in the child feature type. For $x = \text{exists}(C)$, a reference to x could be substituted by `exists(C)`. Substitution of

attributes leads to a denser type declaration as explicit declarations of attributes are allowed to be skipped.

Figure 4.5 illustrates an example FMTD without and with a substituted attribute. The attribute y has been substituted in the constraint of P for the expression that was assigned to y . In both examples the attribute x of an instance of P equals the sum of all z attributes of G instances, children of any C instance. A nested *exists* operation such as $\text{exists}(C.\text{exists}(G))$ returns whether there is a grandchild (G) for any of the children (C) of parent P .

Another example: C has an attribute z and constraint $z = \text{sum}(G.y) > 100$, where P is constrained by $\text{exists}(C.z)$. When z is substituted in P the result for the constraint is $\text{exists}(C.(\text{sum}(G.y) > 100))$. This expresses the constraint that there needs to be a C where the sum of the y attributes of its children (G instances) is larger than 100.

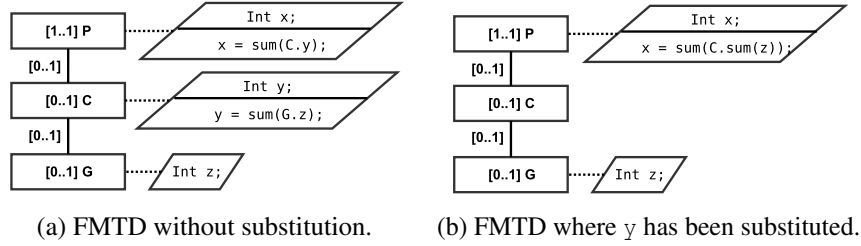


Figure 4.5: Substitution example.

4.2.3 Configurations with Attributes

Attributes are assigned values in configurations. If there is no configuration value for an attribute of an instance, the attribute remains *undecided*. Attribute configurations are declared after an “attributes:” statement, just as in type declarations. An example configuration of *InternetConnection* example (Figure 4.3) is given in Figure 4.6. Different attribute values describe different product configurations, in this line of thoughts *undecided* attributes describe many products. Therefore in for the domain of product specifications (*full* configurations) all attributes need to be configured. Through constraint propagation it is possible to conclude that the *price* attribute is 225 and is added to the configuration of Figure 4.6. As all

attributes have been configured, it is allowed to translate the configuration to a *valid* product specification.

```
1  InternetConnection : InternetConnection1
2    attributes:
3      option = false;
4      connectionName = "Example Connection";
5  group:
6    Wireless : Wireless1
7      attributes:
8        price = 205;
```

Figure 4.6: Example configuration of Figure 4.3.

4.2.4 Mapping Classical Attributed Feature Models to UFTM

The translation of Benavides attributed feature models to UTFM is straightforward. For the mapping we use examples from Figure 4.3.

Attribute Declaration. UTFM requires that attributes are explicitly declared, with a name and a domain. Every attribute that is used in an attributed feature model is therefore declared in a feature type. In the running example, all feature types have two attributes: *DTime* and *Price*, both of type integer.

Constraints. The running example declares each attribute declaration with a range of possible values (*Price* in 100..200). The corresponding arithmetic constraints to specify the allowed range are: $\text{price} \geq 100; \text{price} \leq 200;$. The arithmetic constraint of *Service* is mapped to: $\text{price} = 20 + (\text{sum}(\text{PowerLine.price}) + (\text{sum}(\text{ADSL.price}) + \text{sum}(\text{Wireless.price})))$; The mapping for boolean constraints is already described in Section 2.3.

4.3 Automated Analysis with Z3

The introduction of attributes and arithmetic constraints makes the Z3 model richer. In order to check satisfiability of a model with arithmetic attributes, the theorem prover searches for possible *integer* and *real* values for the attributes if they are not provided in the configuration, in order to satisfy all specified assert statements. Boolean attributes had already been introduced, and correspond to propositional logic in Z3. String attributes have no logic counterpart in Z3, therefore have no influence on the satisfiability of the model. The Z3 translation for boolean and propositional logic has been explained in Section 2.4.1.

4.3.1 Mapping to Z3

Table 4.1 illustrates the mapping of attribute semantics to Z3 from the running example Figure 4.3. As *arithmetic* expressions are defined in the same way as boolean expressions, but with different sub expressions, and different (nested) operations. Besides the three examples in Table 4.1, we provide the Z3 mapping of the different *arithmetic* operations.

The declaration and assert statements for attributes are illustrated in row 1 and 2. Arithmetic asserts use the same structure as boolean asserts in Z3; the function or operation name first, followed by its parameters. In these parameters expressions can be nested. Row 3 of Table 4.1 refers to an example of nested expressions in Z3. The mapping of a *sum* operation consists of nested addition expressions for every instance of the feature type. Each referenced instances' attribute value is multiplied by the *cardinality* function of the instance. Therefore returns 0 when the instance is not selected, otherwise the attribute value. In case the instance is not *selected* nothing is added to the sum; the instance is not part of the configuration and should not add value to the expression.

```
1 (declare-const price Int)
2 (assert (= price 250))
```

Figure 4.7: Z3 arithmetic attribute.

Semantics	FMTD	Configuration	Z3
Arithmetic attribute	Int price;	price = 250;	Figure 4.7
Boolean attribute	Bool option;	option = false;	Figure 4.8
Arithmetic constraints	price = 20 + (sum(Powerline.price) + (sum(ADSL.price) + sum(Wireless.price)));		Figure 4.9

Table 4.1: Examples of mapping from UTFM to Z3.

```

1 (declare-const option Bool)
2 (assert (= option false))

```

Figure 4.8: Z3 boolean attribute.

```

1 (assert (= InternetConnction1.price
2   (+ 20
3     (+ (* (cardinality InternetConnection1.PowerLine1)
4           InternetConnection1.PowerLine1.price)
5     (+ (* (cardinality InternetConnection1.ADSL1)
6           InternetConnection1.ADSL1.price)
7     (* (cardinality InternetConnection1.Wireless1)
8         InternetConnection1.Wireless.price)
9   )
10 )
11 )
12 ))

```

Figure 4.9: Z3 arithmetic constraint.

Max and Min

The *max* and *min* operations are very similar, and both use a Z3 function to translate their semantics to the problem model. The declaration of these functions can be found in Figure 4.10. Both functions take two arithmetic values and return either the maximum of the minimum of the two. The arithmetic values can either be a *attribute reference*, or a nested *arithmetic* operation, or a number value, either `Int` or `Real`. The translation of *average* operations used the same construct as a *sum* operation, however this value is divided by the amount of selected instances, which are counted by the *isSelected* function. An *attribute reference* is specified in a way we’ve already seen in *sum* operations, the attribute value is multiplied by the value of the *isSelected* function. If the attribute is of a selected instance

the attribute value is returned, otherwise it returns 0. Figure 4.11 illustrates the previous mentioned constraints $(\max(\max(C.x), \max(D.x)) \leq 100$; mapped to Z3.

```
1 (define-fun maxAttr ((x Real) (y Real)) Real (ite (< x y) y x))
2 (define-fun minAttr ((x Real) (y Real)) Real (ite (> x y) y x))
```

Figure 4.10: Z3 maximum and minimum functions.

```
1 (assert (<=
2   (maxAttr
3     (maxAttr
4       (* (cardinality P1.C1) P1.C1.x)
5       (* (cardinality P1.C2) P1.C2.x))
6     (maxAttr
7       (* (cardinality P1.D1) P1.D1.x)
8       (* (cardinality P1.D2) P1.D2.x))
9   )
10 100
11 )
```

Figure 4.11: Z3 maximum and average example.

4.3.2 Constraint Propagation

Besides selection values of instances, there is the possibility to derive the value of attributes through constraint propagation. The implementation of constraint propagation for attributes is an extension of the algorithm described in previous Section 2.4.1. The propagation algorithm, based on Janota's algorithm, is extended by also propagating assign expressions. For such expressions it is possible to derive the value from a satisfying model. Based on a given configuration, an value is assigned to a variable in a satisfying model, this value is propagated in the returned configuration of the propagation process. After constraint propagation of configuration file Figure 4.6 the value 225 is propagated to attribute `InternetConnection1.price`. The configuration file that is returned after constraint propagation, provide in Figure 4.12, as a *valid* and *full* configuration. If the input configuration file had declared an instance of a different *InternetConnection* (Powerline or ADSL) the

propagated price would be different.

```
1  InternetConnection : InternetConnection1
2    attributes:
3      price = 225;
4      connectionName = "Example Connection";
5    group:
6      (false) PowerLine : PowerLine82231
7      (false) ADSL : ADSL19283
8      Wireless : Wireless1
9      attributes:
10       price = 205;
```

Figure 4.12: Configuration after constraint propagation of Figure 4.6.

Chapter 5

UTFM Tool

We have implemented the UTFM language as a tool. In this chapter:

1. We explain the functionality of the tool;
2. We show how it is to be used;
3. We provide insight in the structure within the tool.

The source code of the UTFM is published on GitHub, a collaboration and versioning code platform; <https://github.com/vweber/UTFM> . Through the GitHub landing page a pre-compiled windows version of the tool can be downloaded. Installation and compilation instructions are also included on the website. The source code is published under the GNU General Public Licence ¹.

5.1 Functionality and Tool usage

The main purpose of the UTFM tool is to enable the analysis of feature models that are too complex (and labor intense) to validate by hand. Also implementing a compiler for the language and automating the analysis process shows that the proposed theory is feasible. It

¹<http://www.gnu.org/copyleft/gpl.html>

is implemented as a command line tool, that reads and writes UTFM files and is controlled by providing command-line options.

UTFM tool works with FMTD's (uses `utfm` file extension), configurations (`utfmc` file extension) and products (`utfmp` file extension). The full array of command-line options that are supported are explained in Appendix B, and legal combinations of these options are provided in Table 5.1. The tool outputs configuration or product files, or writes errors to the error log file, based on the provided execution options.

#	Use Case	Command Line Execution	Possible output
1	Validating FMTD	<code>utfm -t < FMTD Filepath ></code>	Errors
2	Validating configuration	<code>utfm -t < FMTD Filepath > -c < Configuration Filepath ></code>	Configuration, Errors
3	Validating product	<code>utfm -t < FMTD Filepath > -p < Product Filepath ></code>	Product, Errors
4	Map configuration to product	<code>utfm -t < FMTD Filepath > -c < Configuration Filepath > -f</code>	Product, Errors
5	Propagate configuration	<code>utfm -t < FMTD Filepath > -c < Configuration Filepath > -r</code>	Configuration, Errors
6	Unfold & propagate configuration	<code>utfm -t < FMTD Filepath > -c < Configuration Filepath > -u -r</code>	Configuration, Errors

Table 5.1: Overview of use cases with corresponding UTFM commands and output.

The automated analysis operations that are implemented in the software are: validating type declarations, configurations and products, mapping configurations to products, unfolding configurations and propagating constraints on configurations. We demonstrate the functionality of the UTFM tool through six use cases. An overview of the tool and the different internal processes that are used throughout the use cases are provided in Figure 5.1. The different control flows in the tool are explained in this section. In the next section we look at the implementation of the tool more in depth.

1. Validate FMTD. The automation of validating type declarations. Type declarations need to be sentences of the grammar of all possible type declarations (Appendix A.1).

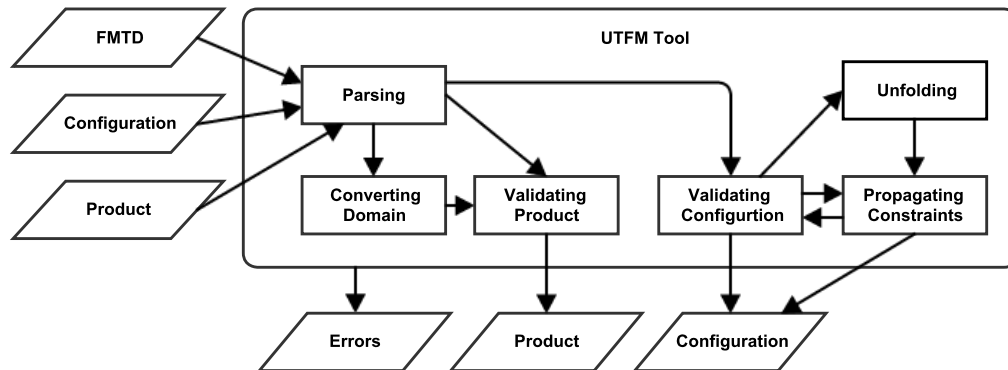


Figure 5.1: Overview on analysis processes in the tool .

Besides that the type declaration is required to be syntactically correct it also needs to satisfy all naming requirements: such as unique names for all child types of a parent and all attributes of a feature type. Furthermore the type declaration is type checked. As the tool only parses the type declaration while not validating it against a configuration (or product), the type declaration is not tested for contradictions (no valid configurations).

Figure 5.2 shows the process of validating a type declaration. The tool requires an FMTD file and returns errors if the type declaration is invalid. Row 1 of Table 5.1 shows how this automated process is to be executed on the command line using the UTFM tool.

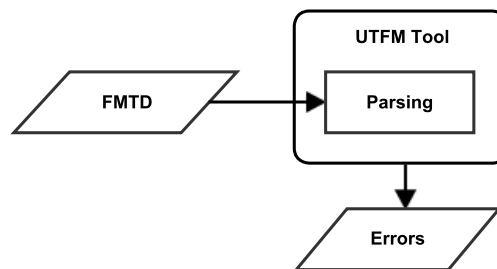


Figure 5.2: Validating type declarations.

2. Validate Configuration. In this process, both type declaration and configuration file are parsed and type checked. A configuration is mapped against a type declaration and is valid if no constraints specified in the FMTD are violated. The actual validation of the feature model and configuration is done by translating both files to Z3 (Section 2.4.1, Section 3.3.1, Section 4.3.1) and checked whether the generated decision problem is satisfiable. If so, the configuration is printed to a separate configuration file. Otherwise, errors are written to an error log file. Figure 5.3 presents the described process in the UTFM tool, row 2 of Table 5.1 the command line execution options.

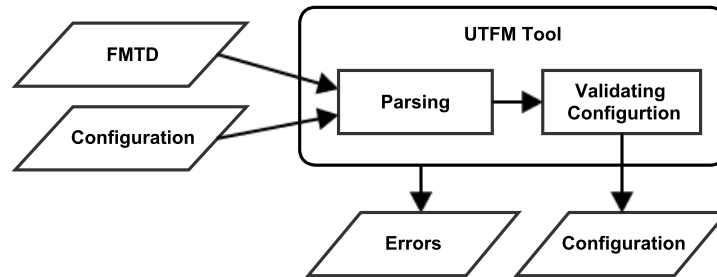


Figure 5.3: Validating configurations.

3. Validate Product. The validation of products is automated as well. Compared to the previous use case a product is mapped against a type declaration instead of a configuration. The Z3 translation of the FMTD and product are checked for satisfiability. If satisfiable the product is pretty printed in a separate file, otherwise errors are written to the error log. Figure 5.4 illustrates this process. Row 3 of Table 5.1 provides the command line.

4. Translate Configuration to Product. The process to translate a configuration to a product requires a type declaration and a configuration file and produces a product file. During the automated process the domain of the configuration is converted to that of products. In products it is only allowed to specify *selected* instances. All *undecided* and *false* configured instances are removed from the configuration and everything that

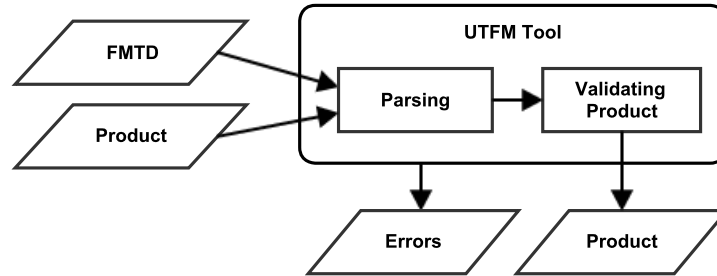


Figure 5.4: Validating products.

is not explicitly *true* is assumed not selected for the product. After the conversion the product is tested for satisfiability. Figure 5.5 shows the described process and row 4 of Table 5.1 the command line execution options.

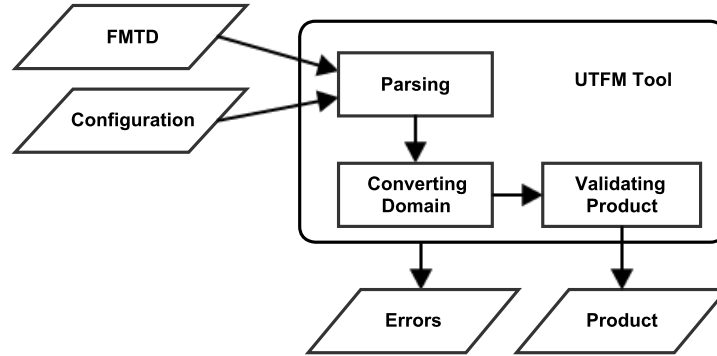


Figure 5.5: Translating configurations to products.

5. Propagate a Configuration. The process of constraint propagation is described in this following use case. First the original configuration file is validated with regard to its type declaration. Second, if the configuration is satisfiable, the propagation algorithm (which we discussed earlier in Section 2.4.1) is invoked. The propagated configuration is printed in a separate configuration file. Figure 5.6 shows the process, row 5 of Table 5.1 the execution options for the UTFM tool.

6. Unfold and propagate a Configuration. By adding an additional option the process de-

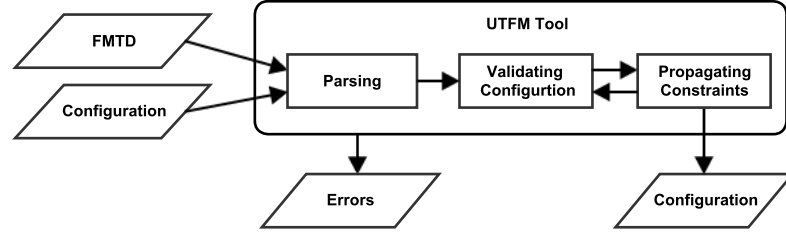


Figure 5.6: Propagating constraints for configurations.

scribed in the previous use case is extended with a unfolding step. After validation of the provided configuration, the configuration is unfolded, up till the upper bound instance cardinality the implicit *undecided* instances are made explicit in the configuration. For all these generated *undecided* instances the constraint propagation algorithm tries to propagate constraints. The process of this use case is described in Figure 5.7, on row 6 of Table 5.1 the command line options for the execution of the UTFM tool are provided.

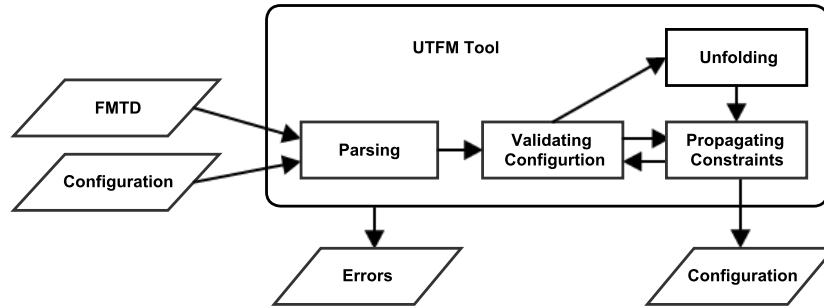


Figure 5.7: Unfolding and propagating constraints for configurations.

5.2 Implementation

The UTFM tool is written in the functional language Haskell. As became clear that UTFM resembled the semantics of an attribute grammar, we explored implementations of such grammars and came across the Haskell implementation of attribute grammars published

by the *University of Utrecht (UU)* ². Advantages of Haskell are that we are able to write dense and powerful algorithms for data manipulation and have increased performance for the analysis processes through lazy evaluation. Therefore we decided to implement the tool using Haskell and the UU attribute grammar libraries. These libraries provide a framework and examples to implement a parser and interpreter that produces and walks an attributed parse tree. Another important external component of the UTFM tool is the Z3 theorem solver ³ that is used to check satisfiability of feature models and configurations.

Figure 5.8 provides an overview of the tool's internal structure. The figure displays how input data traverses to the different internal processes and leads to the output described in the use cases in the previous section. In the tool structure we distinguish two main components the parser and the interpreter.

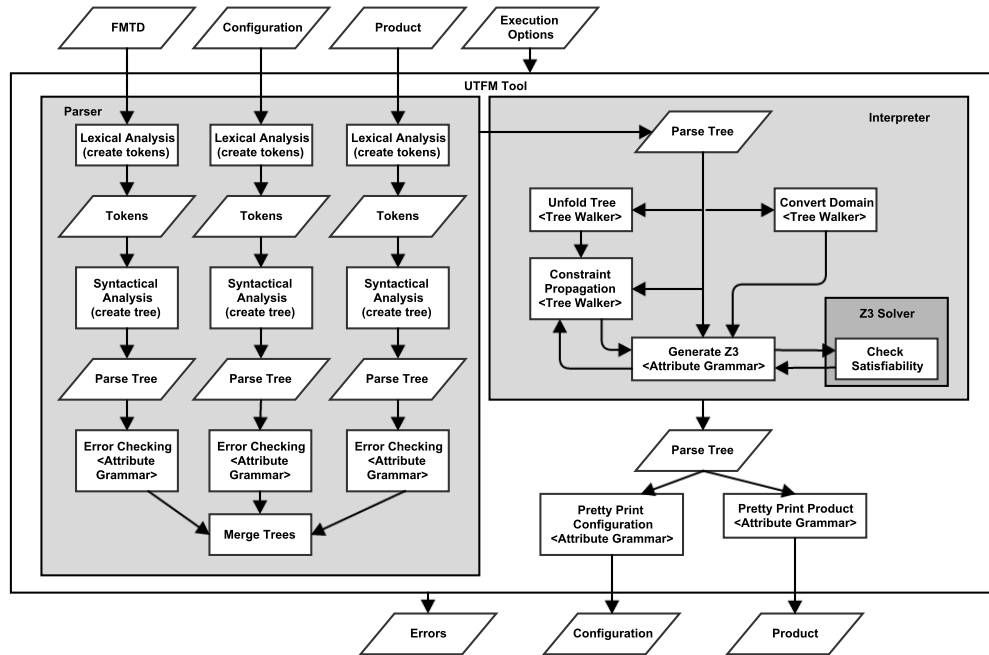


Figure 5.8: Internal structure of the UTFM tool .

The parser performs the lexical and syntactical analysis, followed by error checking.

²<http://www.cs.uu.nl/wiki/HUT/AttributeGrammarSystem>

³<http://z3.codeplex.com/>

As in a typical parser design the language files are first translated to tokens, and eventually put together in a parse tree. Next the parse tree is checked for unique names and the cross-tree constraints are type checked. Finally the parser merges the type declaration with the possible configuration and product parse trees to a single parse tree that is used in the interpreter.

Analysis of parse trees uses two techniques: through the functionality of the attribute grammar and by walking down the tree (a tree walker). Through the inherited and synthesized attributes of the parse tree it is easy to generate different representations of the tree, mainly due to the UUAG pretty print functionality. However it is not possible to change the structure of the tree. For such changes in the structure, for example adding or removing instances in the tree, we need tree walkers. The tree walkers start in the root and walk along the branches of the tree where the algorithm might change the data types. As the structure of the tree changes, the values of the attributes of the attributed tree change as well.

The interpreter component receives a parse tree from the parser, that is assumed a correctly specified UTFM feature model, with possible configuration data. This however does not mean that the configuration or product specification actually represents one or more products. It is still possible that the configuration is *invalid* or that the type declaration contains contradictions in which case there are no *valid* configurations possible.

To check validity, a Z3 representation is generated from the attributed tree and checked for satisfiability with the Z3 solver. Based on the execution options that are provided by the user it is possible for other processes to be called; unfolding, propagating or converting the domain of a configuration to a product domain. These processes are all tree walkers as they change the tree structure. Once interpretation of the parse tree has finished the tool prints a configuration or product file to the file system if the execution options permit this.

Chapter 6

Examples

We demonstrate how the UTFM language could be used in practice.

1. We provide examples that makes use of a combination of feature replication, attributes and complex cross-tree constraints;
2. How the examples describe practical problem domains in which UTFM could be used.

The described software product lines are in the domains: mobile device interfaces and subsea oil production systems (based on work from Behjati et al [6]).

6.1 Mobile Device Interface SPL

Mobile phones and devices come in many shapes, different screen sizes, using tailored software and interfaces. Some devices are operated using touch gestures on the screen, others might use buttons for input. An interface for example can contain several pages through which can be navigated, and all pages contain/present different information. The following examples are about mobile device interfaces, but note the examples provided are only a small subset of possible extensions that could arise in practice. The variability and restrictions that we focus on can be categorized as *content*, *navigation* and *size restrictions*.

Content. In principle anything can be displayed through an interface. It is a matter of detail to which extent it will be specified in the feature model. One can think of possible content such as text, headlines, images, video's and forms. Also the layout of the content is configurable.

Navigation. The possibilities to navigate through the interface affects feature choices through out the type declaration.

Size Restrictions. All the content within the interface needs to be properly displayed, therefore size restriction of the different content elements should hold. These restrictions vary on the types of features that have been configured in the model.

6.1.1 Entities

To get a better understanding of what entities are represented in the model, a list of the entities and their description follows. The descriptions state what the entity represents and how different configurations of the entity could be created for different products. The relations between the different entities are depicted in Figure 6.1.

Window. The `Window` feature is the root feature of the interface, representing the actual window of the mobile device, containing information such as size and resolution and interface settings.

Page. A `Window` can contain multiple `Page`'s which contain the content that is displayed. Through the interface a user can navigate through the different pages.

Navigation. The `Navigation` feature represents the variability possibilities to navigate through the interface, in our example possible options are navigation by gestures on the touch screen or through tabs such as in a browser.

Stylesheet. The `Stylesheet` is used to configure the way the interface is styled, for instance font sizes and background colors of the pages.

GridLayout. The way a `Page` is layout in the interface depends on the `Layout` that is selected. In our current example we only have a `GridLayout` however any other type of layout method can be added. A grid layout can be explained as a three by three matrix where in each cell content can be placed. Through the layout configuration the content is positioned with relation to each other. Figure 6.2a describes such the matrix of a `GridLayout`.

GridObject. Configurations of the `GridObject` feature type represent the actual content displayed in the interface. In our example there are only `Text` and `Image` as types for the content objects. A margin separates objects from one another, in a way that is described in Figure 6.2b.

GridPosition. The position of a object in the `GridLayout` is determined by configuring of two `GridPosition` instances, for instance `Top` and `Left`, or `Center` and `Middle`.

As the type declaration is too large to be displayed in a single figure, Figure 6.1 only shows the features and their group relations, no attributes or constraints are provided. In the following subsections we introduce these attributes and constraints along with to the role they fulfill in the example.

6.1.2 Navigation

An interface on a mobile device can be navigated in different ways, attributes and constraints regarding the specification of navigation variability is provided in Figure 6.3. In this example product line there is the choice between navigating with touch gestures (used in most smartphones) by swiping across the screen or by using tabs (used in internet browsers). This variability is specified in `Navigation` with optional children `GestureNav` and `TabsNav`. In the case that gesture navigation is selected it is required that all `Page`'s have at least one `Gesture` instance selected underneath. `Gesture` is either a swipe right or a swipe left gesture (boolean variable *swipeRight* is *false* is read as swipe left), where *destination* is the

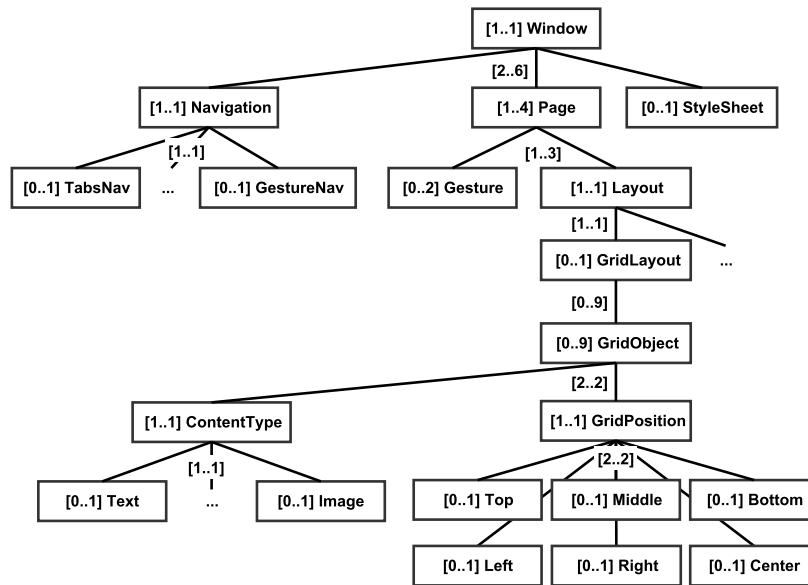
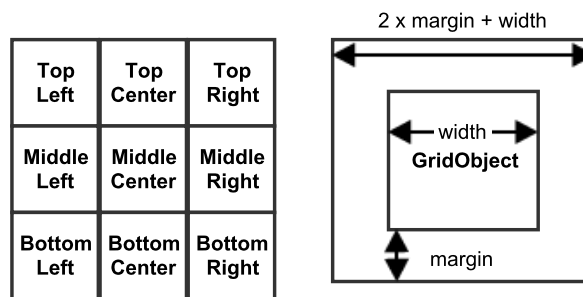


Figure 6.1: Window FMTD.



(a) Possible `GridPosition` combinations. (b) `GridObject` visual interpretation.

Figure 6.2

name of the `Page` that is navigated to when the gesture is made. For the implementation of interfaces that use a navigation through tabs the name attribute of each `Page` could be used.

When content is larger then the size of the screen, the content is displayed by scrolling though it for most devices. Scrolling is usually done by making touch gestures in an up- or downwards direction. In Figure 6.7 attributes and constraints are introduced to

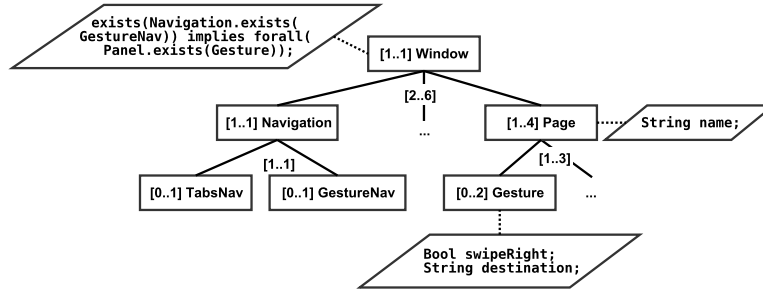


Figure 6.3: Window navigation constraints.

defines the extra constraints put upon the model when scrolling is not possible. The model is constrained that the content can not be taller then the window size, when the *scrollable* attribute is set to *false* in Window. As UTFM permits constraints to be written in various ways Figure 6.7 provides two possible specifications. The constraint of Figure 6.4a is read as: if not *scrollable* then for every Page the *height* should not exceed the *height* of Window, and the constraint of Figure 6.4b is read as: if not *scrollable* then the highest *height* of Page should not exceed the *height* of Window.

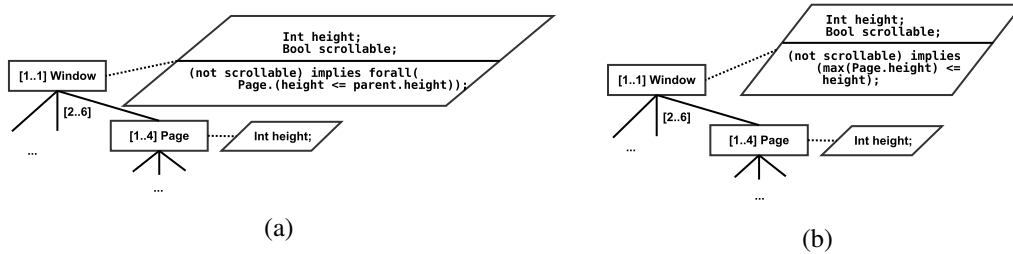


Figure 6.4: Possible scrollability constraints.

6.1.3 Window Size

Whether the constraints are met in the previous subsection is based on the *height* values of the instances of Page. These *height* and *width* values are based on the content that is specified underneath that of their Page; more specifically in the layout that is chosen for the

Page, in our example only GridLayout is provided. Figure 6.5 shows the type declaration with attributes and constraints to determine the window size, additional constraints are given in Figure 6.6 that were too large to display in Figure 6.5.

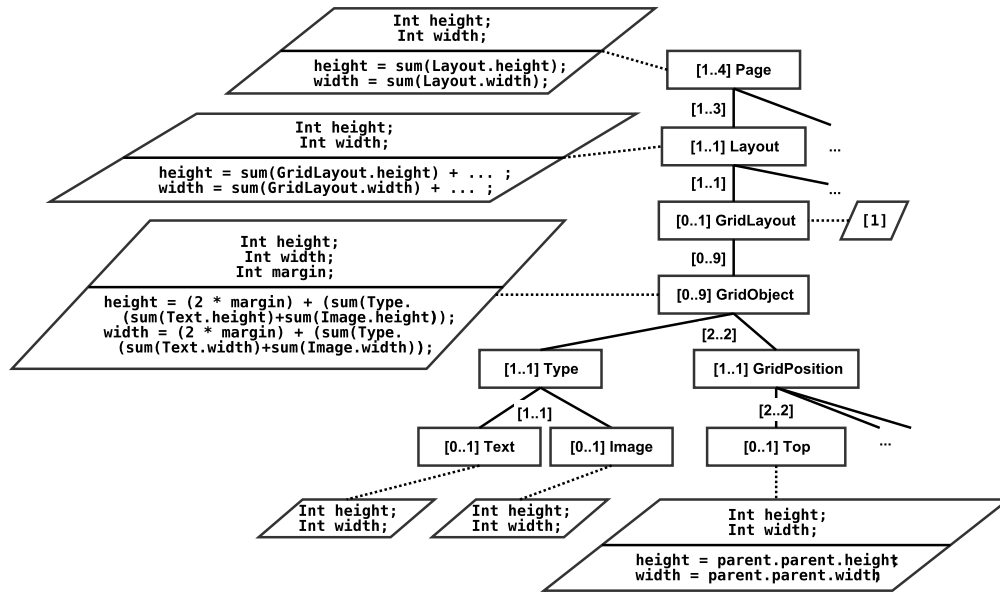


Figure 6.5: Window size constraints.

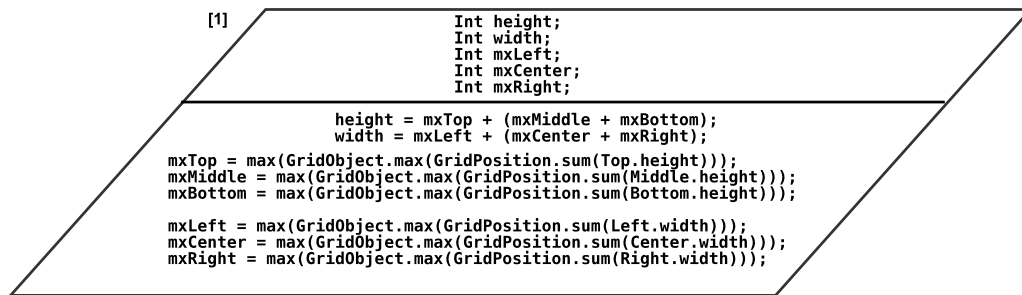


Figure 6.6: Complex constraints to constraint the window size.

Within a **GridLayout** Object's are placed of a certain **Type**, our example is limited to either **Text** or **Image**. For the content objects the dimensions are set and these values will be propagated through the configuration through the constraints. An **GridObject** has

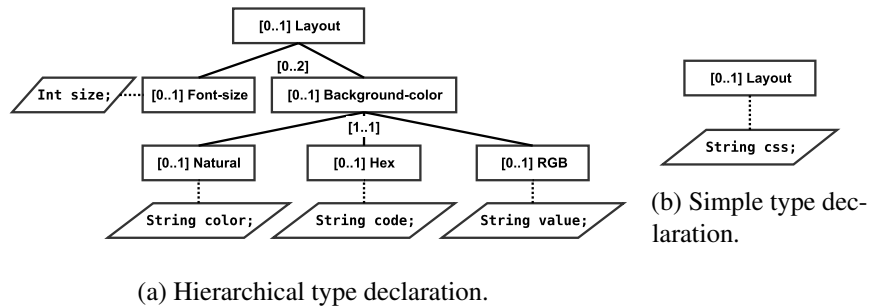


Figure 6.7

a *height* and *width*, and an *margin* that defines the space around an `GridObject`. The size of an `GridObject` is copied to the `GridPosition` children, which is necessary to calculate the maximum size of the grid, in which `GridObject`'s can be stacked above or besides one another.

6.1.4 Stylesheet

Window has a `Stylesheet` feature type that entails the configuration of the visual style of the interface. The variability of visual styles is almost endless considering all the different color combinations, sizes and other visual aspects, for example the language CSS (*Cascading Style Sheets*¹) that is used to write style sheets for websites is very extensive. In our interface example we assume that CSS is used to define style sheets.

There are different ways to express the variability in UTFM, through types declarations and configurations. The first choice is whether the variability of CSS is translated to type declarations or use a single attribute to define the content of the stylesheet. We restrict ourselves to only define the font size of text and the background color. The example interface needs three possible product options, either with a regular font size, a larger font size for visual impaired users, and a smaller font. All options should have a white background, however the color can be defined in CSS in natural language, in a hexadecimal notation and

¹<http://www.w3.org/Style/CSS/>


```

1  (undefined) Stylesheet : Smaller
2    group: {
3      (undefined) FontSize : Fontsize1
4      attributes:
5        size = 11;
6      (undefined) BackgroundColor : BackgroundColor1
7      group: {
8        (undefined) Natural : Natural1
9        attributes:
10         color = "white";
11      }
12    }
13  (undefined) Stylesheet : Regular
14    group: {
15      (undefined) FontSize : Fontsize2
16      attributes:
17        size = 12;
18      (undefined) BackgroundColor : BackgroundColor2
19      group: {
20        (undefined) Hex : Hex1
21        attributes:
22          code = "#FFFFFF";
23      }
24    }
25  (undefined) Stylesheet : Larger
26    group: {
27      (undefined) FontSize : Fontsize3
28      attributes:
29        size = 14;
30      (undefined) BackgroundColor : BackgroundColor3
31      group: {
32        (undefined) RGB : RGB1
33        attributes:
34          value = "RGB (255,255,255) ";
35      }
36    }

```

Figure 6.8: Stylesheet configurations of Figure 6.7a.

in a red, green, blue color combination notation (RGB). Figure 6.7a expresses the variability through hierarchical type declarations, while Figure 6.7b shows Stylesheet with a single attribute for a CSS style sheet.

In the configuration process there will only be one instance selected of the Stylesheet type, however it is possible to define multiple instances, which could be reused in a future configuration process of a different product. In Figure ?? and Figure ?? the three options

```

1 (undefined) Stylesheet : Smaller
2   attributes:
3     body = "font-size: 11pt; background-color: white;";
4 (undefined) Stylesheet : Regular
5   attributes:
6     body = "font-size: 12pt; background-color: #FFFFFF;";
7 (undefined) Stylesheet : Larger
8   attributes:
9     body = "font-size: 14pt; background-color: rgb(255,255,255);";

```

Figure 6.9: Stylesheet configuration of Figure 6.7b.

(regular, larger and smaller) are provided. In the provided figures all instances are *undecided*, however during the configuration of a product, one of the instances could be selected, whereas the others are set to *false* (or left *undecided* and will be propagated to *false* during the propagation process). On the other hand besides these three, a new Stylesheet instance could be declared and selected if there are specific needs for a product.

6.1.5 Tool execution

To execute the Mobile Interface example we use the UTFM command-line tool is the following way: `utfm -t .\var \MobileInterface.utfm -c .\var \MobileInterface.utfmc`

The example type declaration and configuration are provided in the `var` directory of the tool. To view the Z3 decision problem that is generated to check satisfiability, open the temporary `.smt2` file in the `tmp` directory of the tool. For the `MobileInterface` example we see that close to two thousand declare statements are generated and another thousand asserts. As the configuration of the model becomes larger more assert statements will be added to the decision problem.

6.2 Subsea Oil Production SPL

The recent work by Behjati et al. [6] proposes a model-based approach to describe *reference architectures* of SPL's and configure such architectures. A class-like modeling methodology

SimPI is introduced that uses the constructs from Unified Modeling Language (UML2). In their publication it is argued that "feature models are not easily amenable to capturing all kinds of architectural variabilities" and therefore not applicable to their problem domain. The running example, Figure 6.10, that is used to showcase SimPI is in the subsea oil production domain, a product line with an industrial background. As the example is large we only use the fragment of the model presented in figure 3 on page 6 of the publication. The described components in their running example are abstract top-level components, and for their function in the problem domain we refer to their description provided in chapters 2 and 3 [6].

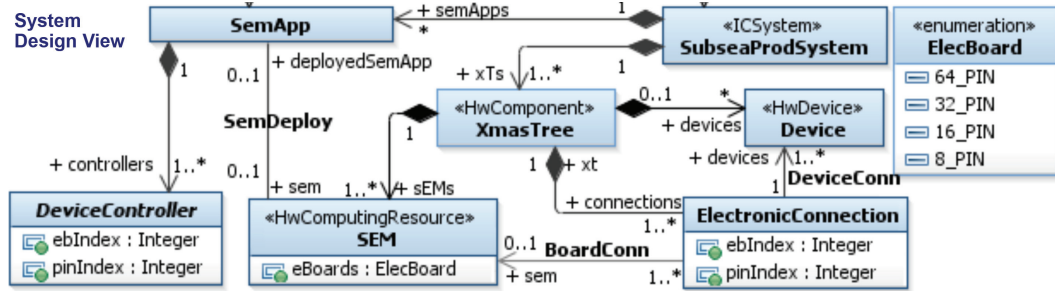


Figure 6.10: The subsea oil production system fragment from [6].

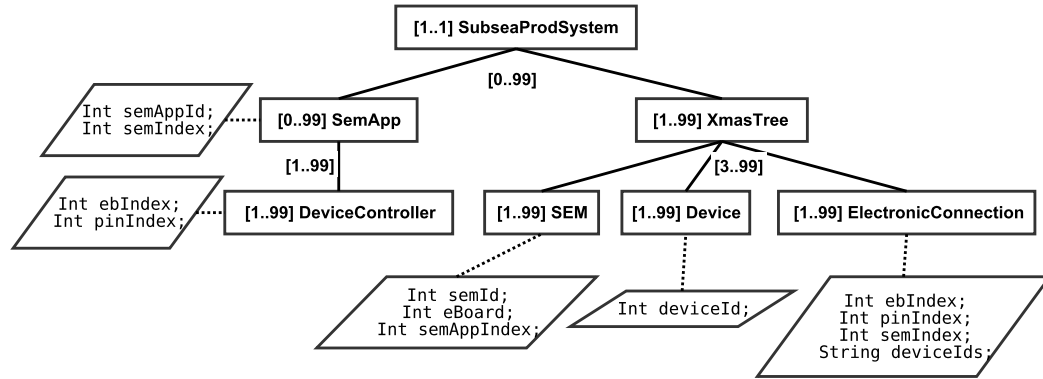


Figure 6.11: UTFM representation of the subsea oil production SPL fragment.

Since we have extended the semantic capabilities of feature models we provided

an reference architecture that entails similar variability as the SimPL model. Figure 6.11 describes the UTFM implementation of the subsea oil production system SPL fragment described in [6]. Nearly all semantics of the SimPL model are translated to the UTFM type declaration, exceptions or notes are listed below regarding: aggregations, associations, enumerations and the Kleene star. The type declaration of Figure 6.11 is provided in Figure 6.12.

```

1  [1..1] SubseaProdSystem
2    group [0..99]:
3      [0..99] SemApp
4        attributes:
5          Int semAppId;
6          Int semIndex;
7        group [1..99]: {
8          [1..99] DeviceController
9            attributes:
10             Int ebIndex;
11             Int pinIndex;
12        }
13    [1..99] XmasTree
14      group [3..99]:
15        [1..99] SEM
16          attributes:
17            Int semId;
18            Int eBoard;
19            Int semAppIndex;
20        [1..99] Device
21          attributes:
22            Int deviceId;
23        [1..99] ElectronicConnection
24          attributes:
25            Int ebIndex;
26            Int pinIndex;
27            Int semIndex;
28            String deviceIds;

```

Figure 6.12: FMTD translation of SimPL fragment in Figure 6.11.

Notes with regard to semantical translations from SimPL (class diagram) to UTFM:

Aggregations. The aggregations between classes are translated to group relations between features. The aggregated class is the child of the parent feature that represents the class that owns the aggregation relation. The multiplicity of the aggregation is translated to

the group cardinality of the group relation.

Associations. Between classes the association relations are translated to feature attributes in UTFM. Each feature that is part of an association relation has an id attribute (Integer), which is used as an index in other features. In the SimPL model there are one-to-one associations and one-to-many. Referencing a single index is done with an Integer attribute in a feature type, referencing many on the other hand is translated to a String attribute; the string contains a list of indexes. In the SimPL model association relations assume that indexes reference id's, this is however not explicitly stated and a direct translation of such constraints in UTFM is not possible. The constraint language does not entail operations to specify such constraints, extra syntax could be added to define foreign keys for associations.

Enumerations. The example uses an enumeration datatype. Hence in most programming languages an enumeration is implemented as an integer, we've translated the enumeration to an integer attribute as well.

Kleene star. Cardinalities in UTFM, in contrast to UML multiplicities, do not permit an infinite upper bound. In multiplicities such an upper bound is described by a Kleene star. In our example we have translate the Kleene star to 99.

The classes in Figure 6.10 are abstractions of more variability then is described in the fragment. Inheritance for features is not supported in UTFM, therefore we refrain our example to just this fragment. With the subsea oil production SPL example we provide evidence that in our proposed language we reduce the gap between feature models and other reference architecture models such as SimPL. The translation is not very polished and requires more elegant solutions for the translation of relations, enumerations, and multiplicities.

Behjati's publication also covers a possible configuration of the SimPl fragment. This configuration is provided in Figure 6.13 (figure 4, page 7 of [6]) and the UTFM translation

in Figure 6.14. In this example we have assumed that all id's for the features have the value 1, and the second entry of the enumeration is represented by the value 2.

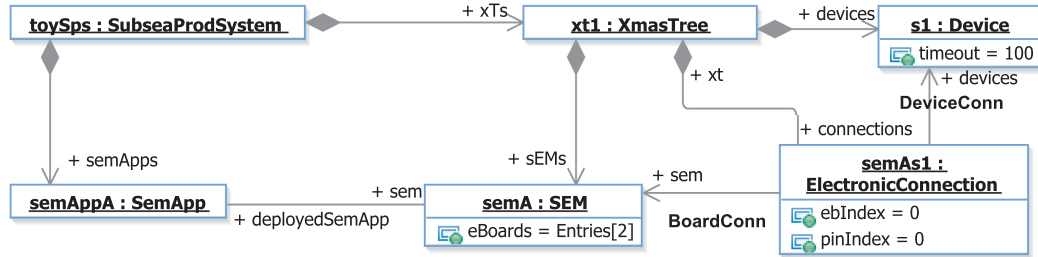


Figure 6.13: Configuration of SimPL fragment Figure 6.10.

```

1  SubseProdSystem:ToySps
2  group:
3    SemApp:SemAppA
4    attributes:
5      semAppId: 1;
6      semIndex: 1;
7  XmasTree:xt1
8  group:
9    SEM:SemA
10   attributes:
11     semId: 1;
12     semAppIndex: 1;
13     eBoard: 2;
14   Device:S1
15   attributes:
16     deviceId: 1;
17   ElectronicConnection:SemAs1
18   attributes:
19     ebIndex: 0;
20     pinIndex: 0;
21     semIndex: 1;
22     deviceIds: "[1]";

```

Figure 6.14: Configuration translation of SimPL configuration in Figure 6.13.

Chapter 7

Related Work

We present the relations of previously published work in regard to the work presented in this thesis.

1. We provide an overview of publications that our work is based on;
2. Furthermore we describe the relation with our work to the research done in parallel to ours.

These topics provide insights into the field and current state of feature modeling and provides readers material for literature review.

Feature modeling was first introduced in 1990 by the work of Kang et al.[23] named *Feature-Oriented Domain Analysis* (FODA), thereafter these initial concepts were refined by Griss et al. in 1998 [17] and Van Gurp et al. in 2001 [36]. In their work they refined the relationships among features. This led to the set of parent-child and cross-tree relationships that are regarded as the classical concepts in feature models that are mapped to UTFM in Chapter 2. Batory published in 2005[4] how basic cross-tree relations can be concatenated with propositional logic in more expressive cross-tree constraints. Furthermore the semantics of feature models are mapped to propositional logic making it possible to perform automated analysis with an off-the-shelf solver. Elements of this mapping and possible concatenation

of constraints are still used in our mapping to Z3.

The introduction of cardinality constraints made an effort to introduce modeling concepts from UML into the languages. Cardinality, a UML-like multiplicity, is introduced by Czarnecki and Eisenecker in 2000 [13] and Riebisch et al. in 2002 [31]. Czarnecki et al. motivate the need for cardinality to describe group relationships in feature models by explaining how they aid in generative programming applications in industry [12]. At a later stage Czarnecki et al. introduce cardinality to describe optionality and the replication of features [14]. In the publication by Michel et al. in 2011 [26] an ambiguity in semantics with regard to nested replicated features is described, which is not addressed in any previous work. UTFM uses cardinality constraints and provides interpretations for nested replication that avoid the ambiguity described by Michel.

The need for additional information in feature models was already expressed in Kang's FODA publication and his 1998 report by Kang et al. [24] where the first form of feature attributes were introduced as non-functional features. In the publications by Benavides et al. [10, 9] and Czarnecki and Kim [15] in 2005 feature attributes were introduced, each a different syntax and level of expressiveness. In Chapter 4 the syntax and semantics of attributes in UTFM are described.

Advancing towards automated analysis feature models were mapped to *satisfiability* (SAT) problems, first described in propositional logic by Batory [4] and Benavides et al. in 2005 [9], and shortly after in *binary search diagrams* (BDD) by Czarnecki and Kim [15] and Benavides et al. [8] However these logical representations are not able to translate arithmetic attributes to a SAT problem, hence can only be part of the solution for automated reasoning for UTFM.

Arithmetic attributes were mapped to *constraint satisfaction problems* (CSP) using constraint programming by Benavides et al. in 2005 [9]. The off-the-shelf solvers that are used to validate CSP's are able to deal with numerical values such as integers and intervals. In another publication by Benavides et al. [8] a comparison is made between the performance

of automated reasoning on feature models using either propositional logic, BDD's or CSP's. *Satisfiability Modulo Theory* (SMT) can be seen as a form of CSP. One of the most recent implemented SMT theorem provers is Z3 from Microsoft Research[16] which is used to validate SMT problems mapped from UTFM.

Another method to perform automated analysis on feature models is through linear programming, where the FM is presented as a mathematical model consisting of linear relationships. Linear programming is a special case of mathematical programming/optimization, which is specifically good in finding optimal solutions in mathematical models. The first mapping of feature models to linear programming problems is published by Van den Broek [35] and uses Integer programming, a extension of linear programming. Another method with regard to linear programming and mathematical optimization is the research from Henneberg that uses pseudo-boolean satisfiability [19] As it is possible to express arithmetic attributes and constraints, linear programming has enough expressiveness to be mapped to from UTFM for automated reasoning.

The following literature reviews shed light on alternative publications, regarding the course of feature modeling research. Schobbens et al. [33] made an overview on the semantics of feature modeling in 2006, Kang et al. [22] provide a similar overview in 2010, two decades after the publication of FODA. Also in 2010 Benavides et al. [7] provided a systematic overview on automated analysis. With regard to the different textual languages and logical representations Classen et al. [11] provided an overview on the extend of implemented semantics in different languages.

Hubaux published in 2012 [20] a work flow process to configure feature models, furthermore an algorithm to detect contradictions while configuring a feature model. The limitations in their work are that feature attributes are only partly implemented and feature replication is not supported. They state that implementation of such, would require a complete adaptation of the semantic domain and significant revision of the constraint system, and is therefore postponed to future work. Our work presents an outline and implementation

of mentioned adaptations and revisions for feature models.

The optimization of configurations for large sized feature models has been discussed in publications by Pohl et al. (2011) [28] and Sayyad et al. (2013) [32]. With regard to the introduction of feature replication in UTFM, resulting in more complex and bigger feature models, research towards more optimized automated analysis is necessary. Interesting in the work by Sayyad is the use of off-the-shelf solver Z3.

The OMG attempt, *Common Variability Language* (CVL)[25], to standardize a variability modeling language has stranded as a proposal in 2012. The language proposed an all-round variability modeling language, besides describing feature models it also abstractly describes the implementation of the product line. The goal of this approach was to bring the problem domain and solution domain of software product lines closer to each other. CVL has been an inspiration for the graphical syntax that has been used throughout our work, while furthermore it provided a starting point to determine which semantics should be part of a next-generation feature modeling language.

Classen et al. [11] proposed in 2011 the text-based feature modeling language *Textual Variability Language* (TVL). The language entails classical and attributed feature models, however does not support feature replication. The interpretation of attributes and constraints is different then from UTFM. Constraints on attributes are evaluated in TVL even if a feature is not selected. Additionally the language was made for rich syntax expression, while UTFM is more strict and consize (using the same language constructs).

The language Clafer, proposed by Bak [2],[3] is a class modeling language with first class support for feature modeling. The models described in Clafer are closer to the implemenation of product lines. Constraints specified in the models are written in Alloy¹. As Clafer uses different semantic concepts there is no mapping from Clafer to feature models provided, making it hard to compare the language to UTFM.

¹<http://alloy.mit.edu>

Chapter 8

Conclusion and Future Work

This thesis is written with the goal to design a feature modeling language with advanced semantics, such as feature attributes, replication and complex constraints, and explore how analysis operations on such models can be automated. The syntax, semantics, and analysis operations of our proposed language UTFM are laid out and illustrated with examples. Furthermore the thesis is accompanied with the UTFM tool that is created to implement the described language and analysis operations.

In this chapter we provide a summary of our work and conclude with some final words with regards to our results. We end this thesis by discussing future endeavors that could follow up on our presented research.

8.1 Results and Conclusion

Feature modeling for software product lines diverged until around 2005 to general accepted standards with regard to syntax and semantics for the modeling languages. As researchers proposed advanced semantics, feature modeling converged in different directions while attempts to unify the different advanced semantic concepts fell short or converged even more from the established classical standards. Besides the introduction of new semantics,

the research for more advanced and powerful off-the-shelf solvers continued creating opportunities for better automated analysis of feature models. In our work we propose the language UTFM to fill the gap in semantic consensus, and describe automated processes for the analysis of such feature models using the Z3 theorem prover.

In the UTFM language a clear distinction between the type declaration and a feature model configuration is made. Feature model type declarations (FMTD) specify feature types, their group relations, attributes and constraints. Classical semantics for features are sustained while features just have an instance cardinality and group relations a group cardinality. This introduced instance cardinality also allows for feature replication. On the other hand in a configuration, instances of feature types are specified along with value assignments for the attributes of these instances. A configuration is valid when no constraints specified in the FMTD are violated.

We elaborate on the semantics of nested feature replication and conclude that group cardinality counts the amount of selected instances in a configuration and that each instance has a local value for its group cardinality. This interpretation of cardinality semantics does not restrict the expressiveness of the language and proves to be the most concise option. The UTFM language supports integer, real, boolean and string feature attributes. Feature types have a local scope which means that constraints can only reference attribute values of themselves, their parents and their children. As constraints reference feature types, instead of instances or instance attribute values, these references are always followed by operations in the constraint language. This constraint language is still dense, as not much syntactic sugar has been added.

For automated validation of the feature model the type declarations and configurations are mapped to a Z3 decision problem. The Z3 theorem solver checks models for satisfiability. If a model is satisfiable, a configuration of the type declaration is valid and represents one or more products. This technique is used to perform automated constraint propagation on configurations. We describe the unfolding algorithm that explicitly defines

default instances to maximize the propagation process. These analysis operations and automated validation of feature models are implemented in a proof-of-concept UTFM tool¹. Finally, examples from in a practical and industrial problem domain are provided to showcase the capabilities of the proposed language.

8.2 Future Work

8.2.1 Reuse and Modularization

When describing larger SPL's the feature models tend to get large and complex hierarchies too. In order to cope with growing complexity and to keep feature models clear, larger models could be split up in several models where one model imports another. An import construct has already been mentioned in the work by Kang et al. in 1998 [24] with the introduction of a layered feature model. *Feature import* (or *feature reference*) semantics could be the first step of the modularization of feature types where features models become reusable objects. For modularizing feature types it might be a good practice to introduce public and private feature attributes, attributes that can be referenced outside of the module or strictly internal attributes. The separation of FMTD's and configurations also raises the question on how modularization affects the configuration of such models.

8.2.2 Optimization

Future work on optimization is two folded. The variability of feature models grows exponentially as more features are introduced to the model. Within the UTFM tool optimizations could lead to better performances for the validation of configurations. A possible optimization is for the set of instances of a feature type to becomes a sorted list. Sorting the list of instances on their selection value, combined with extra Z3 assertions on the satisfiability model that enforces *undecided* instance to behave in a sorted fashion could lead to perfor-

¹<https://github.com/vweber/UTFM>

mance enhancement as there is less redundancy in the model. As we've seen in the related work chapter (Chapter 7) recent publications [32] suggest optimizations for the analysis of feature models, it could be investigated to which extend these optimizations are relevant for our Z3 decision problems.

On the other hand there are opportunities to define optimal values for attributes. In the current UTFM language it is only possible to assign static values to the attributes of instances. Introducing language constructs to search for minimum or maximum values, or other limits, for attributes add to the expressiveness of configuration. Instead of reducing the complexity, search for optimal values increases the complexity of the search problem. In the current stable release of Z3 (4.3.0), value optimization has not been build in, however they are developing this for future releases.

8.2.3 Integer Programming

Currently our type declarations and configurations are translated to Z3 decision problems. However since we've translated the *exists* and *forall* operations for feature replication from a first-order logic problem to a propositional logic problem and further only define arithmetic problems, satisfiability of models could also be checked with integer programming (IP). The first steps on mapping feature models to IP problems have already been made [35]. When such initiatives are extended with a translation of complex constraints to IP problems, quantitative analysis of the performance of UTFM to Z3 or IP problems could be made.

8.2.4 Front-end

The UTFM tool is a command line tool that does nothing more then performing analysis operations. Type declarations, configurations and products need to be specified before execution. To avoid syntactical incorrect UTFM files and define configuration processes a front-end should be developed that uses the UTFM tool as a backbone. A front-end should lead to less errors as a front-end onlu permits to specify legal specifications. Furthermore

as an instance is configured in the front-end, in the background a constraint propagation operation could be run, and present the consequences of the configuration to the user real time.

Appendix A

Context-free Grammars UTFM

A.1 Type declaration

```
1 <Type> ::= <Feature>
2
3 <Feature> ::= <Name>
4     [ "attributes:" <AttributesBlock> ]
5     [ "constraints:" <ConstraintsBlock> ]
6     [ "group" <Cardinality> ":" <FeaturesBlock> ]
7
8 <Features> ::= <Feature>
9     | <Features> <Feature>
10
11 <AttributesBlock> ::= <Attributes>
12     | "{" <Attributes> "}"
13
14 <ConstraintsBlock> ::= <Constraints>
15     | "{" <Constraints> "}"
16
17 <FeaturesBlock> ::= <Features>
18     | "{" <Features> "}"
19
20 <Cardinality> = "[" <Int> ".." <Int> "]"
21
22 <Attribute> ::= "Int" <Name> ";"
23     | "String" <Name> ";"
24     | "Bool" <Name> ";"
25     | "Real" <Name> ";"
26
27 <Attributes> ::= <Attribute>
28     | <Attributes> <Attribute>
29
30 <Constraints> ::= <Constraint>
31     | <Constraints> <Constraint>
```

Figure A.1: Context-free grammar of FMTD's.

A.2 Configuration

```
1 <Configuration> ::= <Instance>
2
3 <Instance> ::= [<Selection>] <TypeName> ":" <Name>
4     [ "attributes:" <AttributesBlock> ]
5     [ "group:" <InstancesBlock>]
6
7 <Instances> ::= <Instance>
8     | <Instances> <Instance>
9
10 <InstancesBlock> ::= <Instances>
11     | "{" <Instances> "}"
12
13 <AttributesBlock> ::= <Attributes>
14     | "{" <Attributes> "}"
15
16 <Attribute> ::= <Name> "=" <Value> ";"
17
18 <Attributes> ::= <Attribute>
19     | <Attributes> <Attribute>
20
21 <Selection> ::= "(true)"
22     | "(false)"
23     | "(undecided)"
24
25 <Value> ::= <Bool> | <Integer> | <Real> | <String>
26
27 <TypeName> ::= <String>
28
29 <Name> ::= <String>
```

Figure A.2: Context-free grammar of Configurations.

A.3 Product

```
1 <Configuration> ::= <Instance>
2
3 <Instance> ::= <TypeName> ":" <Name>
4     [ "attributes:" <AttributesBlock> ]
5     [ "group:" <InstancesBlock>]
6
7 <Instances> ::= <Instance>
8     | <Instances> <Instance>
9
10 <InstancesBlock> ::= <Instances>
11     | "{" <Instances> "}"
12
13 <AttributesBlock> ::= <Attributes>
14     | "{" <Attributes> "}"
15
16 <Attribute> ::= <Name> "=" <Value> ";"
17
18 <Attributes> ::= <Attribute>
19     | <Attributes> <Attribute>
20
21 <Value> ::= <Bool> | <Integer> | <Real> | <String>
22
23 <TypeName> ::= <String>
24
25 <Name> ::= <String>
```

Figure A.3: Context-free grammar of Products.

A.4 Constraint Language

```
1  <Constraint> ::= <BooleanExpr> ";"
2    | <String> "=" <BooleanExpr> ";"
3    | <String> "=" <BooleanProp> ";"
4    | <String> "=" <ArithExpr> ";"
5    | <String> "=" <ArithProp> ";"
6
7  <BooleanExpr> ::= <BooleanFormula>
8    | <BooleanProp>
9    | <BooleanProp> "==" <BooleanProp>
10   | <BooleanProp> "!=" <BooleanProp>
11   | <ArithProp> "==" <ArithProp>
12   | <ArithProp> "!=" <ArithProp>
13   | <ArithProp> "<=" <ArithProp>
14   | <ArithProp> "<" <ArithProp>
15   | <ArithProp> ">=" <ArithProp>
16   | <ArithProp> ">" <ArithProp>
17
18 <BooleanFormula> ::= <BooleanProp> "and" <BooleanProp>
19   | <BooleanProp> "or" <BooleanProp>
20   | <BooleanProp> "implies" <BooleanProp>
21   | "not" <BooleanProp>
22
23 <BooleanProp> ::= <BooleanOp>
24   | <AttributeRef>
25   | <ParentRef>
26   | <Bool>
27   | "(" <BooleanExpr> ")"
28
29 <BooleanOp> ::= "exists(" <FeatureRef> ")"
30   | "exists(" <FeatureRef> "." <BooleanOp> ")"
31   | "exists(" <FeatureRef> "." <AttributeRef> ")"
32   | "exists(" <FeatureRef> "." "(" <BooleanExpr> ")" ")"
33   | "forall(" <FeatureRef> ")"
34   | "forall(" <FeatureRef> "." <BooleanOp> ")"
35   | "forall(" <FeatureRef> "." <AttributeRef> ")"
36   | "forall(" <FeatureRef> "." "(" <BooleanExpr> ")" ")"
```

```

37 <ArithExpr> ::= <ArithProp> "+" <ArithProp>
38   | <ArithProp> "-" <ArithProp>
39   | <ArithProp> "*" <ArithProp>
40   | <ArithProp> "/" <ArithProp>
41
42 <ArithProp> ::= <ArithOp>
43   | <AttributeRef>
44   | <ParentRef>
45   | <Number>
46   | "(" <ArithExpr> ")"
47
48 <ArithOp> ::= "sum(" <FeatureRef> "." <AttributeRef> ")"
49   | "max(" <FeatureRef> "." <AttributeRef> ")"
50   | "max(" <ArithProp> "," <ArithProp> ")"
51   | "min(" <FeatureRef> "." <AttributeRef> ")"
52   | "min(" <ArithProp> "," <ArithProp> ")"
53   | "average(" <FeatureRef> "." <AttributeRef> ")"
54
55 <ParentRef> ::= "parent." <AttributeRef>
56   | "parent." <ParentRef>
57
58 <FeatureRef> ::= <String>
59
60 <AttributeRef> ::= <String>
61
62 <Number> ::= <Integer> | <Real>
63
64 <Bool> ::= "true" | "false"

```

Figure A.4: Context-free grammar of the Constraint Language.

Appendix B

Tool Execution Options

Option	Name	Description
-t <file.utfm>	Type declaration	validate a type declaration, provide the filepath to the FMTD.
-c <file.utfmc>	Configuration	validate a configuration against a type declaration, provide the filepath to the configuration.
-p <file.utfmp>	Product	validate a product against a type declaration, provide the filepath to the product specification.
-u	Unfold	Unfold a configuration.
-r	Constraint Propagation	Propagate constraints on a configuration.
-f	Full configuration	Change the domain of a partial configuration to a full configuration (product).

Table B.1: Execution options UTFM tool.

Bibliography

- [1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer, 2013. page 35.
- [2] Kacper Bak. Clafer: a unified language for class and feature modeling. Technical report, Technical report, Generative Software Development Lab, 2010.
- [3] Kacper Bak, Krzysztof Czarnecki, and Andrzej Wasowski. Feature and meta-models in clafer: mixed, specialized, and coupled. In *Software Language Engineering*, pages 102–122. Springer, 2011.
- [4] Don Batory. Feature models, grammars, and propositional formulas. In *Software Product Lines Conference, Lecture Notes in Computer Sciences*, vol.3714, pages 7–20. Springer, 2005.
- [5] Don Batory, David Benavides, and Antonio Ruiz-Cortés. Automated analysis of feature models: challenges ahead. *Communications of the ACM*, 49(12):45–47, 2006.
- [6] Razieh Behjati, Shiva Nejati, and Lionel C Briand. Architecture-level configuration of large-scale embedded software systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(3):25, 2014.
- [7] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.

- [8] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. A first step towards a framework for the automated analysis of feature models. *Proc. Managing Variability for Software Product Lines: Working With Variability Mechanisms*, pages 39–47, 2006.
- [9] David Benavides, Pablo Trinidad, and Antonio Ruiz Cortés. Using constraint programming to reason on feature models. In *The Seventeenth Conference on Software Engineering and Knowledge Engineering, SEKE 2005*, pages 677–682, 2005.
- [10] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated reasoning on feature models. In *Advanced Information Systems Engineering*, pages 491–503. Springer, 2005.
- [11] Andreas Classen, Quentin Boucher, and Patrick Heymans. A text-based approach to feature modelling: Syntax and semantics of tvl. *Science of Computer Programming*, 76(12):1130–1143, 2011.
- [12] Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, and Ulrich Eisenecker. Generative programming for embedded software: An industrial experience report. In *Generative Programming and Component Engineering*, pages 156–172. Springer, 2002.
- [13] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading, MA, USA, 2000.
- [14] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software process: Improvement and practice*, 10(1):7–29, 2005.
- [15] Krzysztof Czarnecki and Chang Hwan Peter Kim. Cardinality-based feature modeling and constraints: A progress report. In *International Workshop on Software Factories*, pages 16–20, 2005.

- [16] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
- [17] Martin L Griss, John Favaro, and Massimo d’Alessandro. Integrating feature modeling with the rseb. In *Software Reuse, 1998. Proceedings. Fifth International Conference on*, pages 76–85. IEEE, 1998.
- [18] Adithya Hemakumar. Finding contradictions in feature models. In *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings. Second Volume (Workshops)*, pages 183–190, 2008.
- [19] Sebastian Henneberg. Next-generation feature models with pseudo-boolean sat solvers. *Master’s Thesis, University of Passau, Germany*, 2011.
- [20] Arnaud Hubaux et al. Feature-based configuration: Collaborative, dependable, and controlled. *University of Namur, Belgium*, 2012.
- [21] Mikoláš Janota. *SAT solving in interactive configuration*. PhD thesis, University College Dublin, 2010.
- [22] Kyo C Kang. Foda: Twenty years of perspective on feature modeling. In *VaMoS*, page 9, 2010.
- [23] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Technical Report CMU / SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.
- [24] Kyo C Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euseob Shin, and Moonhang Huh. FORM: A feature- oriented reuse method with domain- specific reference architectures. *Annals of Software Engineering*, 5(1):143–168, 1998.

- [25] Common Variability Language. Omg revised submission. <http://www.omgwiki.org/variability/lib/exe/fetch.php?id=start&cache=cache&media=cvl-revised-submission.pdf>, 2012. [Online; accessed 15-May-2014].
- [26] Raphael Michel, Andreas Classen, Arnaud Hubaux, and Quentin Boucher. A formal semantics for feature cardinalities in feature diagrams. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, pages 82–89. ACM, 2011.
- [27] Klaus Pohl, Günter Böckle, and Frank Van Der Linden. Software product line engineering. *Springer*, 10:3–540, 2005.
- [28] Richard Pohl, Kim Lauenroth, and Klaus Pohl. A performance comparison of contemporary algorithmic approaches for automated analysis operations on feature models. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 313–322. IEEE Computer Society, 2011.
- [29] W. V. Quine. Quantification and the empty domain. *The Journal of Symbolic Logic*, 19(3):pp. 177–179, 1954.
- [30] Microsoft Research. Z3 theorem prover. <http://z3.codeplex.com/>, 2014. [Online; accessed 15-May-2014].
- [31] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. Extending feature diagrams with uml multiplicities. In *Proceedings of the Sixth Conference on Integrated Design and Process Technology (IDPT 2002), Pasadena, CA*, 2002.
- [32] Abdel Salam Sayyad, Joseph Ingram, Tim Menzies, and Hany Ammar. Scalable product line configuration: A straw to break the camel’s back. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 465–474. IEEE, 2013.

- [33] P Schobbens, Patrick Heymans, and J-C Trigaux. Feature diagrams: A survey and a formal semantics. In *Requirements Engineering, 14th IEEE international conference*, pages 139–148. IEEE, 2006.
- [34] Steven She and Thorsten Berger. Formal semantics of the kconfig language. *Technical note, University of Waterloo*, page 24, 2010.
- [35] P. M. van den Broek. Optimization of product instantiation using integer programming. In *Proceedings of the 14th International Software Product Line Conference, Volume 2, Jeju Island, South Korea*, pages 107–111, Lancaster, UK, 2010. Lancaster University, Lancaster, UK.
- [36] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. On the notion of variability in software product lines. In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pages 45–54. IEEE, 2001.
- [37] Roman Zippel. Kconfig language. <https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt>, 2009. [Online; accessed 15-May-2014].