MULTI–TARGET IMPLEMENTATION OF A DOMAIN SPECIFIC LANGUAGE FOR EXTENDED FEATURE MODELS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

GÖRKEM DEMİRTAŞ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

DECEMBER 2018

Approval of the thesis:

MULTI–TARGET IMPLEMENTATION OF A DOMAIN SPECIFIC LANGUAGE FOR EXTENDED FEATURE MODELS

submitted by **GÖRKEM DEMİRTAŞ** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Halit Oğuztüzün
Head of Department, **Computer Engineering** _____

Prof. Dr. Halit Oğuztüzün
Supervisor, **Computer Engineering Department, METU** _____

Examining Committee Members:

Prof. Dr. Ali H. Doğru
Computer Engineering Department, METU _____

Prof. Dr. Halit Oğuztüzün
Computer Engineering Department, METU _____

Assist. Prof. Dr. Engin Demir
Computer Engineering Department, Hacettepe University _____

Date: 21.12.2018

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Görkem Demirtaş

Signature:

ABSTRACT

MULTI–TARGET IMPLEMENTATION OF A DOMAIN SPECIFIC LANGUAGE FOR EXTENDED FEATURE MODELS

Demirtaş, Görkem
M.S., Department of Computer Engineering
Supervisor: Prof. Dr. Halit Oğuztüzün

December 2018, 62 pages

Translation of feature models to constraint logic programs is an effective method to enable their automated analysis using existing constraint solvers. More flexibility can be offered for building and application of analysis operations on extended feature models by providing a syntax and mechanism for interfacing the host solver with user defined constraint predicates. These constraints, such as global constraints, can be provided by the constraint solver runtime or by the translator itself as a part of the output. The translator defines a specific parameter passing mechanism for each target environment to be used by the programmer who creates the binding between the translator and the environment. These constraint predicates can use external data sources such as relational databases and application specific algorithms thus separating the concerns of building the model and incorporating domain requirements in analysis steps. In practice such constraints reduce the labeling possibilities for the solver, thereby narrowing down the set of results, i.e. a product's configurations. We describe the design and implementation of an extended feature model compiler supporting syntax for arbitrary predicates, that targets multiple constraint solvers.

Keywords: Domain–Specific Language Implementation, Constraint Logic, Extended Feature Models

ÖZ

GENİŞLETİLMİŞ ÖZELLİK MODELLERİ İÇİN BİR ALANA ÖZGÜ DİLİN ÇOK HEDEFLİ GERÇEKLEŞTİRİMİ

Demirtaş, Görkem
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Halit Oğuztüzün

Aralık 2018 , 62 sayfa

Özellik modellerinin kısıtlı mantık programına dönüştürülmesi, var olan kısıt problemi çözücüler ile bu modellerin otomatik analizi için etkili bir yöntemdir. Hedef kısıtlı mantık programına kullanıcı tanımlı kısıt yüklemleri eklemek için bir sözdizimi ve mekanizma sunulması, model geliştirme ve analiz aşamalarına esneklik kazandırabilir. Bu kısıtlar, örneğin evrensel kısıtlar, kısıt problemi çözücü tarafından veya derleyici çıktısına dahil olarak tanımlanabilir. Derleyici ve hedef çalışma ortamı arasındaki bağlantıyı kuracak programcının kullanımı için derleyici hedefe özgü bir parametre alma mekanizması tanımlar. Bu mekanizma ile tanımlanan kısıtlar, dış veri tabanları ve uygulamaya özgü algoritmalar kullanabilir böylece modelin oluşturulması ve alana bağlı gereksinimlerin analize dahil edilmesi süreçleri birbirinden bağımsız hale getirilebilir. Uygulamada bu kısıtlar, çözüm değişkenlerinin alabileceği değerleri kısıtlayarak sonuç ürün yapılandırma kümesini küçülmesini sağlar. Bu çalışmada, farklı kısıt problemi çözücü ortamlarını destekleyen bir genişletimiş özellik modeli derleyicisi kaynak dil sözdizimi, kullanıcı tanımlı yüklem etkileşim mekanizması ile anlatılmıştır.

Anahtar Kelimeler: Alana Özgü Dil Uygulamaları, Sınırlandırmalı Mantık, Genişletilmiş Özellik Modelleri

To My Mother

ACKNOWLEDGMENTS

I would like to give my thanks to my advisor Halit Oğuztüzün for his unwavering support and encouragement and Ahmet Serkan Karataş for his guidance on theoretical aspects of this work.

This work has been supported by TÜBITAK-ARDEB-1001 program under project 215E188.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface
AST	Abstract Syntax Tree
CLP	Constraint Logic Programming
CSP	Constraint Satisfaction Problem
DSL	Domain Specific Language
EBNF	Extended Backus–Naur Form
IR	Intermediate Representation

CHAPTER 1

INTRODUCTION

1.1 Scope

In this study, a domain specific language (DSL) is defined for describing extended feature models with feature and group cardinalities, feature attributes and arbitrary constraint predicates over features and attributes. A compiler has been developed to translate feature models to constraint satisfaction problems (CSP) for a target constraint solver. The compiler also provides basic feature model analysis operations, several constraint predicates and a mechanism for interfacing the host solver with the feature model itself, for user defined constraint predicates. Optimization, analysis and input–output reporting within the DSL are not covered. Feature model import, export and composition operations are not provided. Complex attribute types are not handled.

1.2 Contributions

The language of the compiler developed is based on the syntax defined in [18] with extensions for declaring signatures for user defined predicates. Such predicates follow a well defined interface between the compiler and its target environments. The grammar has been defined in Lemon ([14]) and Ragel ([26]) syntax. As constraint solver environments, GNU Prolog ([10]), Choco ([23]) and Gecode ([12]) targets have been implemented. One of the important features of the language and its compiler described herein is that they are text based. Thus, a specialized environment is

not needed for development, storage and sharing of the feature models. The implementation source code can be accessed at [13].

Relationships and attributes in a feature model can be thought as a static design which describes a product in terms of broad requirements. As an example, a very fine grained feature model might enumerate all of the possible values of some attribute but it will be syntactically heavy and hard to comprehend or might not be even possible in practice. With syntax support, that attribute can be a parameter of a constraint predicate. Such use allows the designer to declare the domain of the attribute in broad terms and delegate the concern of refining the value to some other source of truth such as a database or an algorithm in an external library.

Another use case of predicate syntax is defining a set of attributes that must have values in relation to each other, which does not have a convenient syntax if not for the constraint predicates. As an obvious example, a predicate can build an SQL query with the supplied attribute references as a row of data to be matched and get the possible values from a database. Administration of such a database can be delegated, for example, to an inventory management system.

While existing syntax allows definition of some constraint on a feature model, it might not have enough performance. A native implementation of the same constraint can be provided to the model by the user in the form of a predicate. Also an external library might be required for a constraint in a model and a predicate which integrates such a library can be provided. Implementation of a predicate is done for a host solver environment once and can be used for any feature model later on.

1.3 Overview

In chapter 2 several concepts such as feature model relations that underlie the domain are explained. In chapter 3, a syntax expressing these concepts and semantic mapping of the constructed model to a CSP are defined. In chapter 4, implementation details such as parsing method and code generation for multiple targets is described. In chapter 5, analysis of the feature model is explained. In chapter 6, an overall summary of development is given and future study topics are proposed.

CHAPTER 2

BACKGROUND

2.1 Domain Specific Languages

Domain specific languages (DSLs) are described in [22] as follows:

> Domain-specific languages (DSLs) are languages tailored to a specific application domain. They offer substantial gains in expressiveness and ease of use compared with general-purpose programming languages in their domain of application.

Specific application domain here is defining feature models through relations between features and constraints over attributes. The compiler implements a free standing DSL ([11]) to generate code for target solver environments that allows executing analysis operations on the feature model.

2.2 Extended Feature Models

Feature models (FM) are a popular notation for defining product line variability and commonality with many variants and extensions described in literature. They represent a set of possible products in terms of hierarchical set of features. They are composed of relations between parent and child features and cross-tree constraints between the branches of the hierarchy ([16],[5]). If the parent feature is not included, its child features are not included in a product as well. Feature relations are briefly;

- Mandatory: Child feature must be included if the parent feature is included in the product.
- Optional: Child feature may be included or not if the parent feature is included.
- Alternative: Within a set of child features, exactly one of them must be included if the parent feature is included.
- Or: Within a set of child features, at least one of them must be included if the parent feature is included.
- Requires: If feature A requires B, both features must be included in a product, if feature A is included.
- Excludes: Given two features, if one of them is included in a product, the other cannot be included.

Further constraints can be defined via logical operators on features which are treated as boolean variables ([18]). In this case, $requires$ can be defined as an implication and $excludes$ can be defined as a negation of an and operation.

An extension to the basic feature model is cardinalities as introduced in ([24],[9]):

- Feature cardinality: Cardinality of a child feature defines the number of its instances that can be included in a product, denoted by a sequence of intervals $[n..m]$ each with a lower bound n and an upper bound m. It is a generalization of the *mandatory* ($[1..1]$) and *optional* ($[0..1]$) relations.
- Group cardinality: An interval $< n - m >$ that defines the number of features from a set of child features that can be included in the product where at least n and at most m features must be selected. It is a generalization of the *alternative* ($< 1 - 1 >$) and *or* ($< 1 - n >$ where n is the size of the set of child features) relations.

Attributes are information about the features and consist of a name, a domain and a value at the minimum ([16],[4],[5]). Similar to the logical constraints on features, arithmetic, relational and logical operators can be used on attributes to define complex

constraints in a model. As a further generalization, propositional logic clauses can be used over a set of attributes ([17]).

2.3 Feature Models as Constraint Programs

A solution to a constraint satisfaction problem is, essentially, assignment of values to a set of variables from their respective domains, simultaneously satisfying a set of propositions over these variables. Aim of translating feature models to CSPs is to use existing constraint solvers such as Gnu Prolog [10] or Choco Solver [23] to find valid product configurations and analyze the model. Further information on constraint logic programming and its application on feature models can be found in [15] and [20]. By using a constraint solver which features variables with integer domains and constraints that operate on integer values, feature models can be mapped to constraint logic programming (CPL) clauses succinctly ([21], [5]). Global constraints can also be used within the model and mapped as described in [17] for more expressive power.

First step of finding solutions to a CSP is declaring *variables* and their *domains*. Next, arithmetic and logical constraints over these variables are defined and stored in a *constraint set*. A *goal* is defined to determine the set of solutions of interest, such as an optimization of some value or simply a valid solution. In the solving phase, propagation and labeling is done repeatedly to find the solutions. *Propagation* is repeatedly reducing the domains of the variables with respect to the constraints on them, until no further removal of values from the domains are possible. *Labeling* is assigning variables a value from their domain without violating any of the constraints. Depending on the goal, labeling continues with a different value from the domain, picked according to the search strategy given while setting up the goal, adding additional constraints if necessary. It continues until no new value can be assigned ([10]). For example, maximizing a certain variable can involve assigning the maximum possible value from its domain and continue labeling other variables. Upon failure, a temporary constraint can be added to reduce the domain of the variable of interest, a backtrack information is stored to roll back the additional constraint and labeling is retried.

2.4 Feature Model Analysis

Here we will mention some of the concepts and analysis operations relevant to our study, explained in detail in [5].

- Full Configuration: It is the complete sets of selected and unselected features. In this study, an assignment of values to all of the attributes are also considered as a part of the full configuration.

- Product: A valid full configuration with respect to the feature model given.

- Partial Configuration: If the union of selected and unselected features doesn't cover all of the features in the model, it is a partial configuration. In this study, a partial assignment of attributes is also considered as a partial configuration, even if all of the features defined in the model is covered by selected and unselected feature sets.

- Valid Configuration: If selected and unselected sets of features don't violate any of the relations declared in the model, it is a valid configuration. Here, we also consider assignments of attributes.

- Filtering: Given a feature model and a configuration, filtering is derivation of products that include the input configuration. For example, given a feature model of a mobile phone, finding all products with WiFi functionality.

- Core features: Set of features that are selected for all of the products.

- Dead features: Set of features that are unselected for all of the products. They often indicate an error in the model.

- Variant features: Set of features that are not in the core or dead feature sets.

- Product count: Given a partial configuration, the count of valid products that include the partial configuration or count of all products that can be derived if no partial configuration is given.

2.5 Related Work

There are several visual and textual tools which implement the extensions to the basic feature models in differing capacities. However they do not support arbitrary constraint predicates as a part of feature model definitions or have limited support. Short descriptions of some relevant tools are given here:

- AHEAD Tool Suite ([3]): It provides feature oriented programming by composing feature implementations in Java. One relevant tool is `guidsl` which takes feature model descriptions and allows building product configurations. The syntax only describes feature relations.
- FAMA Framework ([6]): It is a framework and a visual tool to design and analyze feature models. It can integrate different solvers for analysis. One of the internal feature model representation languages is called AFM which has syntax for attributes and cross tree relations.
- FeatureIDE ([19]): It is a visual editor for feature models which can be exported to several formats. The output can be used for code generation or feature model analysis by other tools. It does not have attribute editing functionality.
- S2T2 ([7]): It is a visual tool for feature model design and basic analysis. It does not handle attributes.
- TVL ([8]): It is a DSL for defining extended feature models. It has syntax for complex types for attribute values and a predefined set of functions. The reference compiler can be used for validity checking and converting a model to a CSP problem in DIMACS format.
- Clafer ([2]): It is a DSL for defining extended feature models which are composable. The syntax contains constructs for optimization goals. There is a suite of tools to visually design feature models, perform analysis and generate product configurations.

- VariaMos ([21]): It is a visual tool for design and analysis of extended feature models. It can generate code for a target constraint solver and has facilities to interface with the target syntax, including predicates. It does not implement a textual DSL.
- Familiar ([1]): It is a highly interactive tool for analysis, modification and composition of feature models, with syntax for import, export, and reporting facilities. It can process various feature model file formats. It does not handle attributes.
- Velvet ([25]): A DSL for describing feature models with syntax for attributes.

The syntax of our DSL resembles TVL ([8]) without scopes and expressiveness is comparable to VariaMos ([21]) which also include predicate support. Our implementation combines the flexibility of the text based tools with the analysis capabilities of graphical tools.

CHAPTER 3

SYNTAX AND SEMANTICS

A new DSL is implemented for extended feature models based on the syntax described in [18]. To include arbitrary predicates in the feature model description, the syntax of predicate parameters includes constructs for;

- Non negative integer and boolean constants.
- Feature references.
- References to existing arbitrary symbols in the host environment.
- Sets of attribute references, grouped by name or feature.
- Neutral values for attributes of unselected features as described in [17].

One important deviation of this implementation from [18] is that, neutral values are always passed instead of elimination of attributes for unselected features as described in [17]. The number of parameters passed to a predicate is determined at compile time and does not change at runtime due to implementation considerations.

3.1 Grammar

Syntax is given in EBNF notation and literal classes in regular expression notation. Comments start with $/\star$ and end with $\star/$ for multiple lines and start with $//$ and end with a newline. Comments are omitted from the grammar given here. Annotations and examples are interleaved to explain some of the details. Line numbers are used for referencing the production rules in the following sections.

Table 3.1: Operator associativity and precedence (lower to higher)

Associativity	Operator
left	`if then`
left	`excludes requires`
left	`iff <->`
left	`implies ->`
left	`or |`
left	`and &`
left	`< > <= >= == !=`
left	`+ -`
left	`/ * mod`
right	`not` - (unary)
left	`. #`
left	`()`

```
<digit> ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
<letter> ::= "_"|"a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"
|"k"|"l"|"m"|"n"|"o"|"p"|"q"|"r"|"s"|"t"|"u"|"v"|"x"|"y"
|"z"|"A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K"|"L"|"M"
|"N"|"O"|"P"|"Q"|"R"|"S"|"T"|"U"|"V"|"X"|"Y"|"Z"
```

```
1. <root>     ::= <rulelist>
2. <rulelist> ::= <rule> ";" { <rule> ";" }
3. <name> ::= <letter> { <letter> | <digit> }
4. <int> ::= <digit> { <digit> }
```

Input consists of statements which are terminated by `;`. Feature relationships are in the top level. `parent ! child` denotes a *mandatory* relation between the `parent` feature and its `child`. Similarly, `parent ? child` is an *optional* relation. A relation with a domain like `parent [1..3] child` denotes a relation with *feature cardinality*. Set of features like `{f1, f2, f3}` is used for the

alternative relation (`!`), *or* relation (`!+`) and relation with *group cardinality* (e.g. `<1-3>`). Group cardinality syntax is more restrictive than full domain syntax such that it doesn't allow multiple intervals.

```
 5. <rule> ::= <adecl>
 6.          | <cexpr>
 7.          | <call>
 8.          | <name> "!" <name>
 9.          | <name> "?" <name>
10.          | <name> "!" <ruledeps>
11.          | <name> "!" "+" <ruledeps>
12.          | <name> <domain> <name>
13.          | <name> "<" <int> "-" <int> ">" <ruledeps>

14. <ruledeps> ::= "{" <siblings> "}"

15. <siblings> ::= <name> { "," <name> }
```

Domains can be a series of non overlapping and increasing intervals of non negative integer values like `[1..8]` and `[2..4][7..9]` or enumeration of increasing non negative integers like `{3,6,9}`.

```
16. <domain_r> ::=
     "[" <int> ".." <int> "]" {"[" <int> ".." <int> "]"}

17. <domainlist> ::= <int> { "," <int> }

18. <domain> ::=  <domain_r>
19.             | "{" <domainlist> "}"
```

A clone feature declared by feature cardinality syntax can be referenced by an indexing such as `f#0`. Indexing is zero based. Clone children of a clone feature in the dependency tree has multiple indexes concatenated from ancestor to descendant. If `p [1..3] a; a [1..2] b;` is given, `b#2#1` is the 2nd clone of `b` under 3rd clone of `a`.

```
20. <idx> ::= <name> "#" <int> { "#" <int> }
```

```
21. <feature> ::= <name>
22.             | <idx>
```

Attributes can be non negative integers or booleans, which are treated as `0` for `false` and `1` for `true` in arithmetic expressions. However, results of relational expressions cannot be used in arithmetic expressions. Operator precedence and associativity (given in table 3.1) is not expressed through the grammar but declared and handled separately in the implementation (chapter 4).

```
23. <attr> ::= <feature> "." <name>
```

```
24. <adecl> ::=   <name> "." <name> "in" <domain>
25.             | <name> "." <name> "boolean"
```

```
26. <iexpr> ::=   <attr>
27.             | <int>
28.             | "true"
29.             | "false"
30.             | "(" <iexpr> ")"
31.             | <iexpr> "+" <iexpr>
32.             | <iexpr> "-" <iexpr>
33.             | "-" <iexpr>
34.             | <iexpr> "/" <iexpr>
35.             | <iexpr> "*" <iexpr>
36.             | <iexpr> "mod" <iexpr>
```

```
37. <bexpr> ::=   <iexpr>
38.             | "(" <bexpr> ")"
39.             | <iexpr> ">" <iexpr>
40.             | <iexpr> "<" <iexpr>
41.             | <iexpr> ">=" <iexpr>
42.             | <iexpr> "<=" <iexpr>
43.             | <iexpr> "==" <iexpr>
44.             | <iexpr> "!=" <iexpr>
```

Cross tree relationships are formed by logical operators.

```
45. <cexpr> ::=   <feature>
46.             | <bexpr>
47.             | "(" <cexpr> ")"
48.             | <feature> "excludes" <feature>
49.             | ( "not" | "~" ) <cexpr>
50.             | <cexpr> ( "and" | "&" ) <cexpr>
51.             | <cexpr> ( "or" | "|" ) <cexpr>
52.             | <feature> "requires" <cexpr>
53.             | <cexpr> ( "implies" | "->" ) <cexpr>
54.             | "if" <cexpr> "then" <cexpr>
55.             | <cexpr> ("iff" | "<->") <cexpr>
```

Built in or user defined constraints can be used in the model. Integer or boolean constants, attributes, features can be given as parameters. Symbols in the host solver environment syntax can also be given as a parameter unchanged like `:symbol` where it will be translated to `symbol` i.e. without the `:`. Parameters like `_.attr` are used to pass the list of attributes which are named `attr` from all features. Parameters like `feature._` is used to pass all attributes of `feature` in the order they are declared. `[` and `]` is used to build a list of variables or a nested list of variables. Nesting level may or may not be significant for a given target. `|` is used to give a default value for attributes or list of attributes whose feature is not selected. Implicit default value is `0`. An example constraint definition for sum of all attributes named `attr` is (with the default value 0 for attributes of the unselected features): `sum_eq(5,_.attr|0)`.

```
56. <paramlist> ::= <param> { "," <param> }

57. <call> ::= <name> "(" <paramlist> ")"

58. <param> ::=   <int>
59.             | "true"
60.             | "false"
61.             | ":" <name>
62.             | <feature>
63.             | <attr> "|" <int>
64.             | <attr>
65.             | "_" "." <name> "|" <int>
66.             | "_" "." <name>
67.             | <feature> "." "_" "|" <int>
68.             | <feature> "." "_"
69.             | "[" <paramlist> "]"
```

3.2 Intermediate Representation

After parsing of the input feature model source, resulting abstract syntax tree (AST) is converted to the intermediate representation (IR) for the code generation step. IR is designed for easy translation of constraints to the declarations in the target solvers, which are GNU prolog CLP clauses and operators ([10]), Choco 4 ([10]) and Gecode 6 ([12]) library functions for constraints.

As the IR is semantically very close to prolog, result of the AST translation described in the following chapters are given in a prolog-like syntax. One important deviation is, operators mentioned are all have CLP semantics and not usual prolog expression semantics. For clarity, a reference of such operators is given in table 3.2.

Table 3.2: IR Operator Descriptions

Operator	Definition
`in ..`	Domain
`=`	Equal
`\=`	Not Equal
`<`	Less
`=<`	Less or Equal
`>`	Greater
`>=`	Greater or Equal
`+`	Addition
`-`	Subtraction
`-` (unary)	Negation
`*`	Multiplication
`/`	Integer Division
`rem`	Remainder
`\/`	Or
`/\`	And
`\+`	Not
`\/\`	Not And
`==>`	Implies
`<=>`	Logical Equivalence

3.3 Syntax Directed Translation

3.3.1 Notation

To describe syntax directed translation of AST, different styles of formatting are used for input, output and value references:

- Token value: Formatted $emphasized$ and can have subscripts. Matches a complete or partial textual value of a token. Textual concatenation is used to denote a new value.
- Production rule reference: Line numbers formatted like ***(1)*** reference the production rule in the grammar given in section 3.1. Two useful sets are defined in the context of the AST, parsed by the referenced rule: $Name_{(i)} = \{name_0, name_1, \ldots\}$ where $name_i$ is the value of the $(i+1)$th `name` terminal parsed by the production rule ***(i)***. Similarly for $Int_{(i)} = \{int_0, int_1, \ldots\}$ where int_i is the value of the $(i+1)$th `int` terminal parsed by the production rule ***(i)***. These sets can be empty. Additionally, $\min(Int_{(i)})$ and $\max(Int_{(i)})$ are defined as the minimum and maximum integer value in that set, respectively.
- Output IR nodes: Formatted with **`bold fixed width`** text
  i.e. **`F_`**$feature_0$ **`==> F_`**$feature_1$ **`,`**.

3.3.2 Variable Sets

In this section, we define several useful sets that are referenced in the following sections, used for building up the body of the main model predicate IR.

3.3.2.1 Feature Tree Construction

Feature declaration is implicit and achieved through relations between a parent and child features. Attributes are declared explicitly, with a domain and an owner feature. Let F the set of all features. Let FT be the set of tuples of parent and child features. As FT is actually a tree of features, it is checked to ensure it is a single acyclic tree.

Let A be the set of tuples of a feature, an attribute, a maximum value, a minimum value, and a set of non negative integer ranges for the holes in the domain. Actions for the production rules:

- ***(8)***,***(9)***,***(12)***: $Name_{(p)} = \{name_0, name_1\} \implies$
  $name_0 \in F, name_1 \in F, (name_0, name_1) \in FT.$

- ***(10)***,***(11)***,***(13)***: For sub-production ***(15)*** let $\forall name_i, name_i \in Name_{(15)}$
  $Name_{(p)} = \{name_0\} \cup Name_{(15)} \implies$
  $name_0 \in F, name_i \in F, (name_0, name_i) \in FT.$

- ***(24)***: For sub-production ***(16)*** let $min = \min(Int_{(16)}), max = \max(Int_{(16)}),$
  $H = \{(int_{2i-1}, int_{2i}) | int_{2i-1}, int_{2i} \in Int_{(16)}, i \in [1, \|Int_{(16)}\|/2)\},$
  for sub-production ***(17)*** let $min = \min(Int_{(17)}), max = \max(Int_{(17)},)$
  $H = \{(int_i, int_{i+1}) | int_i, int_{i+1} \in Int_{(17)}, i \in [0, \|Int_{(17)}\| - 1)\}$
  then $Name_{(24)} = \{name_0, name_1\}$
  $\implies (name_0, name_1, min, max, H) \in A.$

- ***(25)***: $Name_{(25)} = \{name_0, name_1\} \implies (name_0, name_1, 0, 1, \emptyset) \in A.$

3.3.2.2 Feature Clones

Rule ***(12)*** for feature declaration introduces clones of child features recursively. Let FC be the set of tuples of a feature and a number of clones. For sub-productions ***(16)*** or ***(17)*** let $c = max(Int_{(16)}) \vee c = max(Int_{(17)})$ then
$Name_{(12)} = \{name_0, name_1\} \implies (name_1, c) \in FC.$

Let FS be the set of tuples of a feature and a set of names.

- $\forall feature \in F, \forall c \in \mathbb{N}, (feature, c) \notin FC \implies (feature, \{\epsilon\}) \in FS.$

- $\forall (feature, c) \in FC, \forall (parent, feature) \in FT, (parent, S) \in FS \implies$
  $(feature, \{selector\_i | selector \in S, i \in (0, c]\}) \in FS.$

3.3.2.3 Predicate Parameters

Rules ***(63)***, ***(64)***, ***(65)***, ***(66)***, ***(67)***, ***(68)*** introduce a variable which mirrors the value of the referenced attribute if the feature is selected in the model and a neutral value if the feature is not selected. Let V be the set of tuples of a feature, a selector, an attribute and a default value:

- ***(63)***: For sub-production ***(23)***, $Name_{(23)} = \{name_0, name_1\}$,
  $Int_{(23)} = \{int_0, int_1, \ldots\}$,
  $Int_{(63)} = Int_{(23)} \cup \{d\} \implies (name_0, \_int_0\_int_1 \ldots, name_1, d) \in V.$
- ***(64)***: For sub-production ***(23)***, $Name_{(23)} = \{name_0, name_1\}$,
  $Int_{(23)} = \{int_0, int_1, \ldots\} \implies (name_0, \_int_0\_int_1 \ldots, name_1, 0) \in V.$
- ***(65)***: $Name_{(65)} = \{name_0\}, Int_{(65)} = \{int_0\}$,
  $\forall (feature, name_0, a, b, H) \in A, \forall (feature, S) \in FS,$
  $s \in S \implies (feature, s, name_0, int_0) \in V$
- ***(66)***: $Name_{(66)} = \{name_0\}$,
  $\forall (feature, name_0, a, b, H) \in A, \forall (feature, S) \in FS,$
  $s \in S \implies (feature, s, name_0, 0) \in V$
- ***(67)***: $Name_{(67)} = \{name_0\}$, for sub-production ***(21)***, $Int_{(67)} = \{d\}$,
  $\forall (name_0, attr, a, b, H) \in A \implies (name_0, \epsilon, attr, d) \in V.$
  For sub-production ***(22)***, $Int_{(22)} = \{int_0, int_1, \ldots\}, Int_{(67)} = Int_{(22)} \cup \{d\}$,
  $\forall (name_0, attr, a, b, H) \in A \implies (name_0, \_int_0\_int_1 \ldots, attr, d) \in V.$
- ***(68)***: $Name_{(68)} = \{name_0\}$, for sub-production ***(21)***,
  $\forall (name_0, attr, a, b, H) \in A \implies (name_0, \epsilon, attr, 0) \in V.$
  For sub-production ***(22)***, $Int_{(22)} = \{int_0, int_1, \ldots\}$,
  $\forall (name_0, attr, a, b, H) \in A \implies (name_0, \_int_0\_int_1 \ldots, attr, 0) \in V.$

3.3.3 Domain Declarations

Domains are lists of non overlapping non negative integer intervals whose bounds are in increasing order. If the length of the interval list is greater than one, it is a discon-

tinuous domain. Discontinuous domains are modeled as a continuous domain with additional constraints, in order to avoid domain size limitations of implementations in such cases.

Features are mapped to boolean variables. Attribute domains are mapped to non negative integer variables, even if they are declared as boolean in the model. This allows uniform treatment of types in the generated arithmetic and variable list expressions. To avoid inflation of product count when a feature is unselected, its attributes are assigned to the minimum value from their domains.

- For all $(feature, S) \in FS, s \in S$ emitted clauses;
  F_$features$ **in 0..1,**

- For all $(feature, attr, min, max, \{(a, b), \ldots\}) \in A, (feature, S) \in FS,$ $s \in S$ emitted clauses;
  A_$features\_attr$ **in** min**..**max**,**
  \+F_$features$ **==> (A_**$features\_attr$ **=** min**),**
  (A_$features\_attr$ **=<** a**) \/ (A_**$features\_attr$ **>=** b**),...**

- For all $(feature, s, attr, d) \in V, (feature, attr, a, b, H) \in A,$ $min = \min(a, d), max = \max(b, d)$, emitted clauses;
  V_$features\_attr\_d$ **in** min**..**max**,**
  (F_$features$ **/\ V_**$features\_attr\_d$ **= A_**$features\_attr$**) \/**
  (\+F_$features$ **/\ V_**$features\_attr\_d$ **=** d**),**

3.3.4 Feature Relations

3.3.4.1 Mandatory Relations

Mandatory relations are declared by rule ***(8)***.
$Name_{(8)} = \{name_0, name_1\}, (name_0, S) \in FS$, For all $s \in S$, emitted clauses;
F_$name_1 s$ **<=> F_**$name_0 s$**,**

3.3.4.2 Optional Relations

Optional relations are declared by rule ***(9)***.
$Name_{(9)} = \{name_0, name_1\}, (name_0, S) \in FS$, For all $s \in S$, emitted clauses;
F_$name_1 s$ **==> F**_$name_0 s$**,**

3.3.4.3 Feature Cardinality

Relations with feature cardinalities are declared by rule ***(12)***. Let $Name_{(12)} = \{name_0, name_1\}, (name_1, c) \in FC, (name_0, S) \in FS$. Let H be the holes in the cardinality relation domain:

- For sub-production ***(16)*** let $min = \min(Int_{(16)})$,
  $\forall i \in [1, \|Int_{(16)}\|/2), int_{2i-1} \in Int_{(16)}, int_{2i} \in Int_{(16)} \implies$
  $(int_{2i-1}, int_{2i}) \subseteq H$.
- For sub-production ***(17)*** let $min = \min(Int_{(17)})$,
  $\forall i \in [0, \|Int_{(17)}\| - 1), int_i \in Int_{(17)}, int_{i+1} \in Int_{(17)} \implies$
  $(int_i, int_{i+1}) \subseteq H$.

For all $s \in S, i \in [0, c-1)$, emitted clauses;

- If $i + 1 \notin H$; **F**_$name_1 s$_$(i+1)$ **==> F**_$name_1 s$_i**,**.
- If $i + 1 \in H$; **F**_$name_1 s$_$(i+1)$ **<=> F**_$name_1 s$_i**,**.

For all $s \in S$, emitted clauses;

- If $min > 0$; **F**_$name_1 s$_$(min-1)$ **<=> F**_$name_0 s$**,**.
- If $min = 0$; **F**_$name_1 s$**_0 ==> F**_$name_0 s$**,**.

3.3.4.4 Group Cardinality

Rules ***(10)***, ***(11)***, ***(13)*** declare relations with group cardinality where domain of the number of selected child features is given explicitly or implicitly.

- ***(13)***: Let $Name_{(13)} = \{name_0, name_1, name_2, \ldots\}, Int_{(13)} = \{int_0, int_1\}$, $int_0 \leq int_1 \leq \|Name_{(13)}\| - 1, (name_0, S) \in FS$, Let u be an unique identifier. For all $s \in S$ emitted clauses;
  G_u **in 0..**int_1**,**
  G_u **= F_**$name_1 s$**+F_**$name_2 s$**+...,**
  (F_$name_0 s$ **/\ G_**u **>=** int_0**) \/ (\+F_**$name_0 s$ **/\ G_**u **= 0),**

- ***(11)***: Let $Name_{(11)} = \{name_0, name_1, name_2, \ldots\}, c = \|Name_{(11)}\| - 1$, $(name_0, S) \in FS$, Let u be an unique identifier. For all $s \in S$ emitted clauses;
  G_u **in 0..**c**,**
  G_u **= F_**$name_1 s$**+F_**$name_2 s$**+...,**
  (F_$name_0 s$ **/\ G_**u **>=** 1**) \/ (\+F_**$name_0 s$ **/\ G_**u **= 0),**

- ***(10)***: Let $Name_{(10)} = \{name_0, name_1, name_2, \ldots\}, (name_0, S) \in FS$. For all $s \in S$ emitted clauses;
  F_$name_0 s$ **= F_**$name_1 s$**+F_**$name_2 s$**+...,**

3.3.5 Predicates

The main interface of the feature model to the host environment is predicate calls. They are predefined but the compiler has no knowledge of the arity of the predicate or the correct types of the parameters. Let $P_{(i)} = \{p_0, p_1, \ldots\}$ be the set of IR sub-nodes generated by the production rule ***(i)*** where p_j is the $(j+1)$th sub-node generated:

- ***(57)***: $Name_{(57)} = \{name_0, \ldots\}$ and for sub-production ***(56)***, $P_{(56)} = \{p_0, p_1, \ldots\}$ emitted clauses; $name_0$ **(**p_0**,** p_1**, ...),**
- ***(58)***: $Int_{(58)} = \{int_0\}$, emitted clauses; int_0
- ***(59)***: Emitted clauses; **1**
- ***(60)***: Emitted clauses; **0**
- ***(61)***: $Name_{(61)} = \{name_0\}$, emitted clauses; $name_0$

- ***(62)***: $Name_{(62)} = \{name_0\}$, for sub-production ***(21)***, emitted clauses;
  `F_`$name_0$,
  for sub-production ***(22)***, $Int_{(22)} = \{int_0, int_1, \ldots\}$, emitted clauses;
  `F_`$name_0$`_`int_0`_`int_1`...`

- ***(63)***: For sub-production ***(23)***, $Name_{(23)} = \{name_0, name_1\}$,
  $Int_{(23)} = \{int_0, int_1, \ldots\}, Int_{(63)} = Int_{(23)} \cup \{d\}$ emitted clauses;
  `V_`$name_0$`_`int_0`_`int_1`..._`$name_1$`_d`

- ***(64)***: For sub-production ***(23)***, $Name_{(23)} = \{name_0, name_1\}$,
  $Int_{(23)} = \{int_0, int_1, \ldots\}$ emitted clauses;
  `V_`$name_0$`_`int_0`_`int_1`..._`$name_1$`_0`

- ***(65)***: $Name_{(65)} = \{name_0\}, Int_{(65)} = \{int_0\}$,
  $\forall(feature, name_0, a, b, H) \in A, \forall(feature, S) \in FS$,
  $s \in S \implies$ `V_`$features\_name_0\_int_0 \in P_{(65)}$.
  For $P_{(65)} = \{p_0, p_1, \ldots\}$ emitted clauses; `[`p_0`,`p_1`,...]`

- ***(66)***: $Name_{(66)} = \{name_0\}$,
  $\forall(feature, name_0, a, b, H) \in A, \forall(feature, S) \in FS$,
  $s \in S \implies$ `V_`$features\_name_0$`_0`$\in P_{(66)}$.
  For $P_{(66)} = \{p_0, p_1, \ldots\}$ emitted clauses; `[`p_0`,`p_1`,...]`

- ***(67)***: $Name_{(67)} = \{name_0\}$, for sub-production ***(21)***, $Int_{(67)} = \{d\}$,
  $\forall(name_0, attr, a, b, H) \in A \implies$ `V_`$name_0\_attr\_d \in P_{(67)}$.
  For sub-production ***(22)***, $Int_{(22)} = \{int_0, int_1, \ldots\}, Int_{(67)} = Int_{(22)} \cup \{d\}$,
  $\forall(name_0, attr, a, b, H) \in A \implies$ `V_`$name_0$`_`int_0`_`int_1`..._`$attr\_d \in P_{(67)}$.
  For $P_{(67)} = \{p_0, p_1, \ldots\}$ emitted clauses; `[`p_0`,`p_1`,...]`

- ***(68)***: $Name_{(68)} = \{name_0\}$, for sub-production ***(21)***,
  $\forall(name_0, attr, a, b, H) \in A \implies$ `V_`$name_0$`_`$attr$`_0` $\in P_{(68)}$.
  For sub-production ***(22)***, $Int_{(22)} = \{int_0, int_1, \ldots\}, Int_{(68)} = Int_{(22)}$,
  $\forall(name_0, attr, a, b, H) \in A \implies$ `V_`$name_0$`_`int_0`_`int_1`..._`$attr$`_0` $\in P_{(68)}$.
  For $P_{(68)} = \{p_0, p_1, \ldots\}$ emitted clauses; `[`p_0`,`p_1`,...]`

- ***(69)***: For sub-production ***(56)***, $P_{(56)} = \{p_0, p_1, \ldots\}$
  emitted clauses; `[`p_0`,`p_1`,...]`

3.3.6 Expressions

3.3.6.1 Literal And Attribute Expressions

Production rules ***(26)*** to ***(44)*** defines relations between attribute values. Let $E_{(i)} = \{e_l, e_r\}$ be the set of IR sub-nodes generated by the production rule ***(i)*** where e_l IR sub-node generated by the production rule before the operator and e_r the sub-node generated by the production rule after the operator. Only e_l may exist for unary or implicit operations. Semantics of the generated operators are given in table 3.2:

- ***(26)***: $Name_{(26)} = \{name_0, name_1\}$,
  $Int_{(26)} = \{int_0, int_1, \ldots\}$ emitted clauses;
  A_$name_0$_int_0_int_1**...**_$name_1$
- ***(27)***: $Int_{(27)} = \{int_0\}$, emitted clauses; int_0
- ***(28)***: Emitted clauses; **1**
- ***(29)***: Emitted clauses; **0**
- ***(30)***: $E_{(30)} = \{e_l\}$, emitted clauses; e_l
- ***(31)***: $E_{(31)} = \{e_l, e_r\}$, emitted clauses; **(**e_l **+** e_r**)**
- ***(32)***: $E_{(32)} = \{e_l, e_r\}$, emitted clauses; **(**e_l **-** e_r**)**
- ***(33)***: $E_{(33)} = \{e_l\}$, emitted clauses; **(-**e_l**)**
- ***(34)***: $E_{(34)} = \{e_l, e_r\}$, emitted clauses; **(**e_l **/** e_r**)**
- ***(35)***: $E_{(35)} = \{e_l, e_r\}$, emitted clauses; **(**e_l ***** e_r**)**
- ***(36)***: $E_{(36)} = \{e_l, e_r\}$, emitted clauses; **(**e_l **rem** e_r**)**
- ***(37)***: $E_{(37)} = \{e_l\}$, emitted clauses; **(**e_l **> 0)**
- ***(38)***: $E_{(38)} = \{e_l\}$, emitted clauses; e_l
- ***(39)***: $E_{(39)} = \{e_l, e_r\}$, emitted clauses; **(**e_l **>** e_r**)**
- ***(40)***: $E_{(40)} = \{e_l, e_r\}$, emitted clauses; **(**e_l **<** e_r**)**

- ***(41)***: $E_{(41)} = \{e_l, e_r\}$, emitted clauses; $(e_l$ `>=` $e_r)$
- ***(42)***: $E_{(42)} = \{e_l, e_r\}$, emitted clauses; $(e_l$ `=<` $e_r)$
- ***(43)***: $E_{(43)} = \{e_l, e_r\}$, emitted clauses; $(e_l$ `=` $e_r)$
- ***(44)***: $E_{(44)} = \{e_l, e_r\}$, emitted clauses; $(e_l$ `\=` $e_r)$

3.3.6.2 Cross-Tree Relations

Production rules ***(45)*** to ***(55)*** defines relations between features which are not necessarily parent and child. Let $C_{(i)} = \{c_l, c_r\}$ be the set of IR sub-nodes generated by the production rule ***(i)*** where c_l IR sub-node generated by the production rule before the operator and c_r the sub-node generated by the production rule after the operator. Only c_l may exist for unary or implicit operations. Semantics of the generated operators are given in table 3.2:

- ***(45)***: $Name_{(62)} = \{name_0\}$, for sub-production ***(21)***, emitted clauses;
  `F_`$name_0$
  For sub-production ***(22)***, $Int_{(22)} = \{int_0, int_1, \ldots\}$, emitted clauses;
  `F_`$name_0$`_`int_0`_`int_1`...`
- ***(46)***: Let $F = \{f_0, f_1, \ldots\}$ be the set of feature variable IR sub-nodes generated and for each production rule ***(26)*** let $Name_{(26)} = \{name_0, name_1\}$, $Int_{(26)} = \{int_0, int_1, \ldots\} \implies$ `F_`$name_0$`_`int_0`_`int_1`... `$\in F$. Let e be the IR sub-node generated by non-terminal `<bexpr>` then emitted clauses;
  $(e$ `/\` f_0 `/\` f_1 `/\` `...)`
- ***(47)***: $C_{(47)} = \{c_l\}$, emitted clauses; c_l
- ***(48)***: $C_{(48)} = \{c_l, c_r\}$, emitted clauses; $(c_l$ `\/\` $c_r)$
- ***(49)***: $C_{(49)} = \{c_l\}$, emitted clauses; `(\+`c_l`)`
- ***(50)***: $C_{(50)} = \{c_l, c_r\}$, emitted clauses; $(c_l$ `/\` $c_r)$
- ***(51)***: $C_{(51)} = \{c_l, c_r\}$, emitted clauses; $(c_l$ `\/` $c_r)$
- ***(52)***: $C_{(52)} = \{c_l, c_r\}$, emitted clauses; $(c_l$ `==>` $c_r)$

- ***(53)***: $C_{(53)} = \{c_l, c_r\}$, emitted clauses; **(**c_l **==>** c_r**)**
- ***(54)***: $C_{(54)} = \{c_l, c_r\}$, emitted clauses; **(**c_l **==>** c_r**)**
- ***(55)***: $C_{(55)} = \{c_l, c_r\}$, emitted clauses; **(**c_l **<=>** c_r**)**

CHAPTER 4

COMPILER ORGANIZATION

4.1 Implementation

The compiler is implemented in C using Lemon [14] parser generator and Ragel [26] state machine compiler as the lexer. It receives a feature model source and optionally a template and generates Gnu Prolog, Java 9 (for Choco solver version 4 [23]) or C++11 (for Gecode version 6 [12]) source code. General structure of the implementation is shown in figure 4.1. Compiler source, build and running instructions are given in Appendix B.

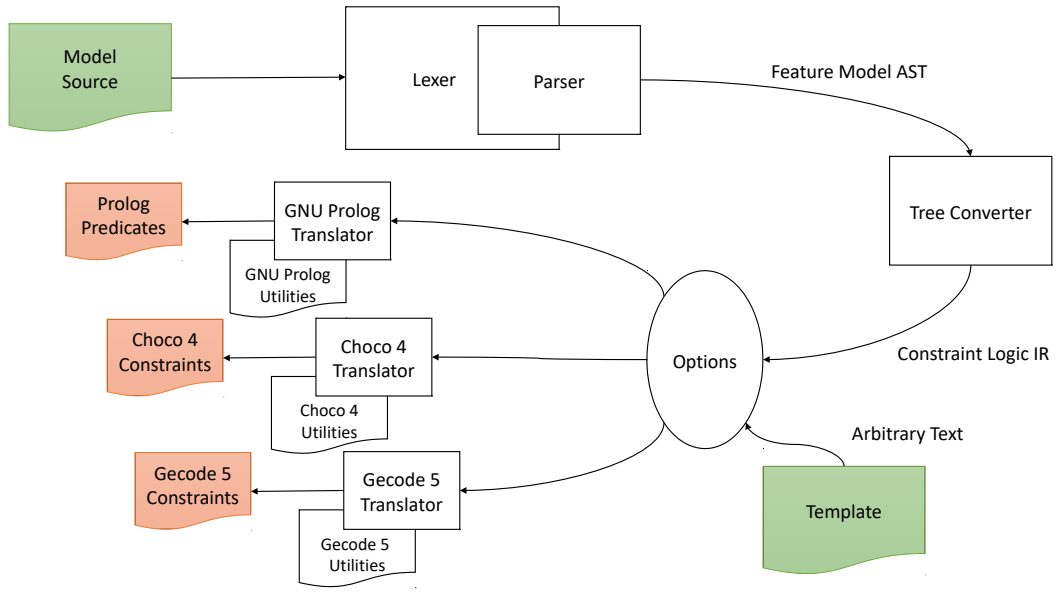

Figure 4.1: Compiler Block Diagram

4.2 Utility Global Constraints

There are a few useful global constraints defined by the compiler for all of the targets. More constraints can be defined by the user in a similar fashion for a specific target. As parameters to the constraints, boolean attributes and features are treated as `1` and `0` for true and false boolean attributes respectively, selected and unselected for features respectively.

- `sum_eq(S,V)`: Let `V` be a list of parameters, then `S` is the sum of these parameters.
- `min_eq(M,V)`: Let `V` be a list of parameters, then `M` is the minimum of these parameters.
- `max_eq(M,V)`: Let `V` be a list of parameters, then `M` is the maximum of these parameters.
- `all_different(V)`: Let `V` be a list of parameters, then they are constrained to be different from each other.
- `all_different_0(V)`: Let `V` be a list of parameters, then they are constrained to be different from each other if they are not `0`.

4.3 Output Code Generation

The generated IR consists of named variables and clauses for the feature model (FM). In this section outputs for the targets are described by giving examples for clarity. General structure of the output is:

- FM object declaration.
- Declaration FM variables and their domains.
- Initialization of FM variables.
- Declaration FM Constraints.

- Constraint solving strategy setup.
- Declaration of helper objects for analysis operations.

Example input models and excerpts from their outputs is given in Appendix A. FM object outputs for the targets are explained here.

Actual analysis step is not handled in the generated code as it requires a specification for analysis queries and resulting output, which is not a part of the DSL. Instead, analysis objects are generated to be used with the host environment facilities. Their use is explained in chapter 5. In analysis, for all targets, features and attributes are referenced by the same name as in the model source. Names of clone features are concatenated by their indexes with _, i.e. `f#0#4` becomes `f_0_4`. Generated code for different targets suit different uses cases best:

- For development of the feature model, prolog target can be used which is highly interactive and has good debugging facilities.
- For configurators, Choco (java) target can be integrated into a graphical user interface with databases for constraints and analysis options.
- For other automated tools, Gecode (C++) target offers best performance but lacks flexibility of the other targets.

4.3.1 Prolog Target

The main FM object consists of generated predicates named `product_model/3`, `product_hook/3` and `product_descriptor/4` but these are not the intended public interface. The public interface is the analysis predicates and the predicate `product_configuration(C)` where `C` is a list of items which describe the order of features and attributes to construct a complete or partial product configuration that can be used with other predicates. For the example in section A.1 an example query may yield;

```
?- product_configuration(C).
C = [root,f1,f2-[attr1-_,attr2-_],f3_0,f3_1,f3_2,f3_3]
```

Use of this result for FM analysis is explained in section 5.1. User defined predicates in the model can be arbitrary prolog predicates. If these predicates are backtracking, they may instantiate a part of the FM multiple times for the same solution. To avoid it, all of the predicates are collected and run in `product_hook/3`.

4.3.2 Choco 4 Target

The main FM object consists of `Product` class with `features` and `attributes` properties, which are modeled as inner classes with properties named after features and attributes. User predicates are modeled as protected methods of the `Product` class, accepting `IntVar` parameters and returning `Constraint` objects. Analysis operations are in `Analysis` inner class, explained in more detail in section 5.2. Example output excerpt can be seen in section A.3. Package name, model class name and the class body can be changed by providing a custom template to the compiler.

4.3.3 Gecode 6 Target

The main FM object consists of a template class `Model` which a concrete class inherits from. The first parameter of the template should be the inheriting class itself. Optionally, other mix-in classes can be given as parameters. The purpose of the recurring template parameter and mix-in classes as parameters is to enforce correct resolution of method overrides.

The `Model` class has `features` and `attributes` properties, which are modeled as inner classes with properties named after features and attributes. User predicates are modeled as public methods of the `Model` class, accepting `IntVar` or `IntVarArgs&` parameters and handle adding constraints to the model, themselves. Analysis operations are methods in `Analysis` inner class, explained in more detail in section 5.3. Example output excerpt and template instantiation can be seen in section A.4.

CHAPTER 5

MODEL ANALYSIS

Implemented DSL itself does not provide any analysis constructs, as explained before. However, as a part of the generated output, following analysis operations defined in section 2.4 are intended to be provided along with a full or partial configuration (as a *filtering* operation) and additional cross-tree constraints:

- Finding valid products.
- Counting valid products.
- Core features.
- Dead features.
- Variant features.

For object oriented targets, this list is not completely implemented but inheritance mechanism and the host application programming interface (API) can still be used. Similarly, more involved types of analysis operations are not provided, such as optimization, for they are not generic enough to generate code from a common IR for all targets. A complete analysis example by using prolog output is given in Appendix C.

5.1 Prolog Target

The output consists of `product_configuration/1`, `product_valid/3`, `product_count/3`, `features_core/3`, `features_optional/3`, `features_dead/3` predicates built with CLP expressions and related predicates:

5.1.1 Product Description

`product_configuration(C):`
`C` is a list of items which describe the order of features and attributes to construct a complete or partial product configuration that can be used with other analysis predicates. An example query may yield;

```
?- product_configuration(C).
C=[root,f1,f2-[attr1-_,attr2-_],f3]
```

From this output, an example partial product configuration can be constructed;

```
PC=[+root,-f1,f2-[attr1-30|_],_]
```

where a feature name with + indicates it is selected and with − indicates it is unselected in the configuration. In the example `f1` is not selected and `f3` is not specified. Features with attributes does not have + but has the attribute list present when they are selected, as `f2` in the example. When declaring a partial product, feature and attribute order is significant.

5.1.2 Valid Configuration and Filtering

`product_valid(Product,Filters,Goal):`
`Product` is a product configuration whose structure is described in subsection 5.1.1 if there are any valid products found. `Filters` is a list of expressions to filter configurations by selected and unselected features and bind attributes to variables. Unlike product configuration, the ordering is not important. It has its own syntax:

- `+feature`: An atom with unary + operator defines that the feature with the same name is selected.
- `-feature`: An atom with unary − operator defines that the feature with the same name is unselected.

- `f-attr=X`: Let `F` be the boolean value that represents whether the feature `f` is selected or unselected in the configuration and `A` the value of its attribute `attr`. Then `X` is unified with `F-A`.

- `+f-attr=X`: Attribute named `attr` of feature name `f` is unified with `X`. Feature `f` is also selected in the configuration.

- `f-[]=X`: Let `F` be the boolean value that represents whether the feature `f` is selected or unselected in the configuration and `AL` the value of its attribute list. Then `X` is unified with `F-AL`. Attribute order is significant.

- `+f-[]=X`: Attribute list of feature name `f` is unified with `X`. Attribute order is significant. Feature `f` is also selected in the configuration.

- `attr=X`: All attributes named `attr` are made a list and unified with `X`. Attributes of unselected features are set to `0` in `X`.

- `attr^D=X`: All attributes named `attr` are made a list and unified with `X`. Attributes of unselected features are set to `D` in `X`.

`Goal` is the condition to further filter by just before labeling features and attributes. It can use the `X` variables unified in the filters. It is called as `call(Goal)`.

An example query is;

```
?- product_valid(P,[+f2-[]=[A,6],+f1,-f3,f1-attr=B,
cost=C],A#>5).
```

In this query, `f2` should be selected and its attributes should match the given list where `A` is bound to the value of first declared attribute and the second one equals `6`. `f1` should be selected, `f3` should be unselected. Let the value of `attr` of `f1` be `X` and selected value of `f1` be `S`, then `B` is `S-X`. Configurations are further filtered by `A#>5`. All attributes named `cost` are collected in the list `C`.

5.1.3 Product Count

`product_count(Count,Filters,Goal):`
`Count` is the number of valid configurations filtered by `Filters` and `Goal`. `Filters` and `Goal` are the same as in subsection 5.1.2. An example query is;

```
?- product_count(C,[+f1],true).
```

5.1.4 Feature Occurrences

`features_core(Core,Filters,Goal),`
`features_dead(Dead,Filters,Goal),`
`features_optional(Opt,Filters,Goal):`
`Core` is the list of features which are selected in all products. `Dead` is the list of features which are not selected in all products. `Opt` is the list of features which are both selected and unselected in non empty subsets of products. All these predicates return the results filtered by `Filters` and `Goal` which are defined the same as in subsection 5.1.2.

5.2 Choco 4 Target

The output FM class has an inner class named `Analysis` with methods to do basic analysis operations on the FM instance. Optimization and complex analysis can be done using the FM object directly.

5.2.1 Filtering

The constructor of `Analysis` class can optionally take `ReExpression` objects as the constraints for all of the subsequent analysis operations. These constraints can be used to define a partial configuration. For the simple FM in section A.1:

```
Product p=new Product();
Product.Analysis a=p.new Analysis(
                          p.attributes.f2.attr1.eq(3));
```

For checking validity, `boolean check(ReExpression... expr)` method can be used, optionally with additional constraints: `a.check()==true;`

5.2.2 Product Count

Method `long count()` returns the count of valid products: `a.count()==8;`

5.2.3 Processing Valid Products

Method `void valid(IMonitorSolution callback)` executes the given callback object for every valid configuration found:

```
a.valid((IMonitorSolution)()->{
    System.out.printf("%d ",p.features.f3_3.getValue());
})
```

```
1 1 0 0 0 0 1 1
```

5.2.4 Feature Occurrences

`Map<String,IntVar> featuresCore()`, `Map<String,IntVar> featuresDead()` and `Map<String,IntVar> featuresOptional()` methods return the dictionary of feature variables keyed by their names which are core, dead and optional features respectively:

```
a.featuresCore().keySet()
```

```
[root, f2, f3_0, f3_1]
```

5.3 Gecode 6 Target

The traditional development environments of C++ language is not conductive to interactive programming thus no obvious approach to performing model analysis exists. An easy to use API specification for analysis is left as an open problem. In this study, analysis in C++ target is not fully implemented therefore the generated `Analysis` inner class is more of an example than a ready to use interface. `int count()` method demonstrates how to initialize the search for all of the solutions and access each solution. Output example is given in section A.4.

CHAPTER 6

CONCLUSION

6.1 Achievements

A textual DSL for extended feature models with syntax for constraint predicates is defined and a compiler to convert them to CLPs is implemented. Such predicates can be used for defining global constraints and external constraints that can be used to decouple non-deterministic aspects of the product line from the design of the feature model, such as inventory management and production capabilities. The predicate syntax can express the necessary connection of these aspects inside the feature model. The implementation is intended to be independent of specialized tools as much as possible, to be easy to learn, use and integrate with other tools. The compiler is extended to target multiple constraint solver environments. Users can pick the one best suited to their needs, abilities and environments. For each target, an API for user provided constraints is defined. Utility constraints are provided by using the same mechanism. Helper objects are included into the output for FM analysis.

For the domain, design of the AST became a balancing act of simplifying expression constructs and synthesizing nodes, for correct handling of attribute and feature interaction in expressions and correct value handling for the host environment. IR design had a sizable impact on the ease of development of the multiple target code generators. It had to be generic enough while allowing the use of optimized constructs in the host.

6.2 Future Work

Syntax of the DSL could be extended to cover more use cases. There is no syntax for optimization goals or analysis queries as in [6]. Such operations are delegated to the host solver environment with a set of generated helper objects. Including such constructs into the DSL might help analysis that works in multiple environments. Syntax for operations on models such as inclusion of external models, merging or extraction as in [1] is not supported. If included, it can improve model source code modularity and allow reuse. String or data structure support for attributes as in [8] is not provided, since all attributes are treated as non negative integer values to allow a uniform mechanism of parameter passing for constraint predicates. Support for more types can improve expressiveness.

Implementation of the compiler itself can be improved. Error reporting of the compiler can be improved by storing more information, such as line numbers, in the tokens and keeping track of first encounters and repeat errors. More constraint solver targets can be added.

REFERENCES

[1] M. Acher, P. Collet, P. Lahire, and R. B. France. Familiar: A domain-specific language for large scale management of feature models. *Science of Computer Programming*, 78(6):657–681, 2013.

[2] K. Bąk, K. Czarnecki, and A. Wąsowski. Feature and meta-models in clafer: mixed, specialized, and coupled. *SLE*, pages 102–122, 2010.

[3] D. Batory. Feature-oriented programming and the ahead tool suite. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 702–703. IEEE, 2004.

[4] D. Benavides, P. T. Martín-Arroyo, and A. R. Cortés. Automated reasoning on feature models. In *CAiSE*, volume 5, pages 491–503. Springer, 2005.

[5] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.

[6] D. Benavides, S. Segura, P. Trinidad, and A. R. Cortés. Fama: Tooling a framework for the automated analysis of feature models. *VaMoS*, 2007:01, 2007.

[7] G. Botterweck, M. Janota, and D. Schneeweiss. A design of a configurable feature model configurator. 2009.

[8] Q. Boucher, A. Classen, P. Faber, and P. Heymans. Introducing tvl, a text-based feature modelling language. In *Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10), Linz, Austria, January*, pages 27–29, 2010.

[9] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software process: Improvement and practice*, 10(1):7–29, 2005.

[10] D. Diaz, S. Abreu, and P. Codognet. On the implementation of gnu prolog. *Theory and Practice of Logic Programming*, 12(1-2):253–282, 2012.

[11] M. Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.

[12] Gecode Team. Gecode: Generic constraint development environment, 2017. `http://www.gecode.org` Accessed Tue, 20 Mar. 2018.

[13] Görkem Demirtaş. Multitarget dsl for extended feature models, 2018. `https://github.com/e1448596/fmdsl` Accessed Tue, 20 Mar. 2018.

[14] D. R. Hipp. *Lemon Parser Generator*. Hipp, Wyrick & Company, Inc., 2017. `https://www.hwaci.com/sw/lemon/index.html` Accessed Tue, 20 Mar. 2018.

[15] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *The journal of logic programming*, 19:503–581, 1994.

[16] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

[17] A. Karataş, H. Oğuztüzün, and A. Doğru. Global constraints on feature models. *Principles and Practice of Constraint Programming–CP 2010*, pages 537–551, 2010.

[18] A. S. Karataş, H. Oğuztüzün, and A. Doğru. From extended feature models to constraint logic programming. *Science of Computer Programming*, 78(12):2295–2312, 2013.

[19] C. Kastner, T. Thum, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel. Featureide: A tool framework for feature-oriented software development. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 611–614. IEEE, 2009.

[20] M. Mannion. Using first-order logic for product line model validation. *Software Product Lines*, pages 149–202, 2002.

[21] R. Mazo, C. Salinesi, D. Diaz, and A. Lora-Michiels. Transforming attribute and clone-enabled feature models into constraint programs over finite domains. In *6th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, 2011.

[22] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, Dec. 2005.

[23] C. Prud'homme, J.-G. Fages, and X. Lorca. *Choco Documentation*. TASC,LS2N, CNRS UMR 6241 and COSLING S.A.S., 2017. `http://www.choco-solver.org` Accessed Tue, 20 Mar. 2018.

[24] M. Riebisch, K. Böllert, D. Streitferdt, and I. Philippow. Extending feature diagrams with uml multiplicities. In *6th World Conference on Integrated Design & Process Technology (IDPT2002)*, volume 23, 2002.

[25] R. Schröter, T. Thüm, N. Siegmund, and G. Saake. Automated analysis of dependent feature models. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, page 9. ACM, 2013.

[26] A. Thurston. *Ragel FSM Compiler*. Colm Networks, 2017. `http://www.colm.net/open-source/ragel/` Accessed Tue, 20 Mar. 2018.

APPENDIX A

EXAMPLE MODELS

A.1 Simple Model

Input source;

```
root ? f1;
root ! f2;
root {2,4} f3;

f2.attr1 in [1..100];
f2.attr2 boolean;
```

A.2 Prolog Output Excerpt

```
product_model(F_root,
    [F_root, F_f1, F_f2, F_f3_0, F_f3_1, F_f3_2, F_f3_3],
    [A_f2_attr1, A_f2_attr2]
):-
fd_domain_bool(F_root),
fd_domain_bool(F_f1),
fd_domain_bool(F_f2),
fd_domain_bool(F_f3_0),
fd_domain_bool(F_f3_1),
fd_domain_bool(F_f3_2),
fd_domain_bool(F_f3_3),
fd_domain(A_f2_attr1,1,100),
(( #\ F_f2) #==> (A_f2_attr1 #= 1)),
fd_domain(A_f2_attr2,0,1),
(( #\ F_f2) #==> (A_f2_attr2 #= 0)),
(F_f1 #==> F_root),
(F_f2 #<=> F_root),
(F_f3_3 #<=> F_f3_2),
```

```
(F_f3_2 #==> F_f3_1),
(F_f3_1 #<=> F_f3_0),
(F_root #<=> F_f3_1).
...
product_descriptor(
    [F_root, F_f1, F_f2, F_f3_0, F_f3_1, F_f3_2, F_f3_3],
    [A_f2_attr1, A_f2_attr2],
    [
        root - F_root,
        f1 - F_f1,
        f2 - F_f2,
        f3_0 - F_f3_0,
        f3_1 - F_f3_1,
        f3_2 - F_f3_2,
        f3_3 - F_f3_3
    ],[
        f2 - [
            attr1 - A_f2_attr1,
            attr2 - A_f2_attr2
        ]
    ]
):-
(F_root #>0 ).
...
product_valid(Product,Filters,Goal):-
    ...
product_count(Count,Filters,Goal):-
    ...
features_core(Core,Filters,Goal):-
    ...
features_dead(Dead,Filters,Goal):-
    ...
features_optional(Opt,Filters,Goal):-
    ...
```

A.3 Java Output Excerpt

```
package product;

import org.chocosolver.solver.Model;
import org.chocosolver.solver.variables.IntVar;
import org.chocosolver.solver.variables.BoolVar;
import org.chocosolver.solver.constraints.Constraint;
import org.chocosolver.solver.search.loop.monitors.IMonitorSolution;
import org.chocosolver.solver.expression.discrete.relational.ReExpression;

public class Product extends Model{

    public class Features{
        public BoolVar root=...
        public BoolVar f1=...
        public BoolVar f2=...
        public BoolVar f3_0=...
        public BoolVar f3_1=...
        public BoolVar f3_2=...
        public BoolVar f3_3=...
    }
    public Features features=new Features();
    public class Attributes{
        public class Attr_f2{
            public IntVar attr1=...
            public IntVar attr2=...
        }
        public Attr_f2 f2=new Attr_f2();
    }
    public Attributes attributes=new Attributes();
    ...
    protected Constraint sum_eq(IntVar s,IntVar[] vars){ ...
    ...
    public class Analysis{
        public Analysis(ReExpression... filters){ ...
        public Map<String,IntVar> featuresCore(){ ...
        public Map<String,IntVar> featuresOptional(){ ...
        public Map<String,IntVar> featuresDead(){ ...
        public long count(){ ...
        public void valid(IMonitorSolution callback){ ...
        public boolean check(ReExpression... expr){  ...
    }
}
```

A.4 C++ Output Excerpt

```
#include <gecode/minimodel.hh>
#include <gecode/search.hh>
using namespace Gecode;
template <typename T,typename ...M> class Model:public M...,public Space{
    friend T;
public:
    ...
    void sum_eq(IntVar s,IntVarArgs& a){ ...

    class Analysis{
    ...
    public:
        Analysis(Model* model){
            ...
        int count(){
            int solutions=0;
            DFS<Model> search(m);
            Model* solution;
            while(solution=search.next()){
                ++solutions;
                delete solution;
            }
            return solutions;
        }
    };

    struct Features{
        BoolVar root;
        BoolVar f1;
        BoolVar f2;
        BoolVar f3_0; BoolVar f3_1; BoolVar f3_2; BoolVar f3_3;
        ...
    }; Features features;

    struct Attributes{
        struct Attr_f2{
            IntVar attr1;
            IntVar attr2;
            ...
        }; Attr_f2 f2;
        ...
    }; Attributes attributes;
    ...
};
```

C++ model usage example;

```
class Product:public Model<Product>{};

int main(int argc,char** argv){
    Product p;
    Product::Analysis a(&p);
    cout<<a.count()<<endl;
    return 0;
}
```

A.5 Bike Model

```
bike ! frame;
bike [2..2] wheel;
bike ! drivetrain;
bike [2..2] brake;
bike ! seat;
bike ? accessories;

bike.total_cost in [1..100000]; bike.total_weight in [1..30000];
sum_eq(bike.total_cost,_.cost); sum_eq(bike.total_weight,_.weight);

wheel ! rim; wheel ! spokes; wheel ! hub; wheel ! {tire,solidtire};

hub ? brakedisc;
tire ! tube;

drivetrain ! {chain,belt};
drivetrain [0..2] shifter;
drivetrain ! crankset;
drivetrain ! {sprockets,pulley};
drivetrain ? tensioner;

crankset ! {chainrings,crankpulley};
shifter ? derailleur;
seat ! seatpost; seat ! saddle;

/* Material
    1 steel
    2 aluminium
    3 cabon fiber
*/
frame.material in [1..3];
frame.hasdisc boolean;
```

```
frame.weight in [4000..8000];
frame.cost in [100..1000];

rim.material in [2..3];
rim.diameter in {16,20,24,26};
rim.holes in {24,32,36};
rim.weight in [100..300];
rim.cost in [30..300];

spokes.length in [1..20];
spokes.count in [0..100];

/* Hub type
    1 front hub
    2 dynamo hub
    3 freehub
    4 gear hub
*/
hub.type in [1..4];
hub.diameter in [5..10];
hub.holes in {24,32,36};
hub.hasdisc boolean;
hub.gears in {0,3,7};
hub.spline in [0..2][7..10];
hub.weight in [50..1000];
hub.cost in [30..300];

tire.diameter in {16,20,24,26};
tire.weight in [80..150];
tire.cost in [20..40];

tube.diameter in {16,20,24,26};
solidtire.diameter in {16,20,24,26};
solidtire.weight in [150..300];
solidtire.cost in [280..300];

chain.weight in [150..150];
chain.cost in [100..100];

belt.weight in [50..50];
belt.cost in [400..400];

shifter.gears in [2..10];
shifter.cost in [30..100];

seatpost.material in [1..3];
seatpost.weight in [100..300];
```

```
seatpost.cost in [50..150];

saddle.weight in [500..1000];
saddle.cost in [100..300];

/* Brake type
    1 rim brake
    2 disc brake
*/
brake.type in [1..2];
brake.weight in [25..100];
brake.cost in [50..200];

chainrings.count in [1..3];

/* Derailleur type:
    1 front
    2 rear
*/
derailleur.type in [1..2];
derailleur.gears in [2..10];

sprockets.count in [1..10];
sprockets.cost in [50..200];

rim#0.diameter==rim#1.diameter;

rim#0.holes==hub#0.holes;
rim#0.holes==spokes#0.count;
rim#0.material==3 -> brake#0.type!=2;
hub#0.type==1 or hub#0.type==2;
brake#0.type==2 -> frame.hasdisc and hub#0.hasdisc;
(rim#0.diameter-hub#0.diameter)/2+1==spokes#0.length;
rim#0.diameter==tire#0.diameter;
tire#0.diameter==tube#0.diameter;

rim#1.holes==hub#1.holes;
rim#1.holes==spokes#1.count;
rim#1.material==3 -> not brake#1.type!=2;
hub#1.type==3 or hub#1.type==4;
brake#1.type==2 -> frame.hasdisc and hub#1.hasdisc;
(rim#1.diameter-hub#1.diameter)/2+1==spokes#1.length;
rim#1.diameter==tire#1.diameter;
tire#1.diameter==tube#1.diameter;

shifter#0 requires shifter#0.gears==chainrings.count or
    shifter#0.gears==hub#1.gears;
```

```
shifter#1 requires shifter#1.gears==sprockets.count;

chainrings.count>1 -> derailleur#0.type==1;
sprockets.count>1 -> derailleur#1.type==2;

sprockets.count>7 -> chainrings.count<3;
chainrings.count>sprockets.count -> hub#1.gears>0;

derailleur#0 requires shifter#0.gears==derailleur#0.gears and hub#1.gears==0;
derailleur#1 requires shifter#1.gears==derailleur#1.gears;
derailleur#1 requires tensioner;

chain requires sprockets and chainrings;
belt excludes derailleur#0;
belt excludes derailleur#1;
belt requires pulley and crankpulley;

sprockets requires hub#1.spline>=sprockets.count;
hub#0.spline==0;

frame.material==3 <-> seatpost.material==3;

//inventory
inventory(:frame,frame._);
inventory(:brake,brake#0._);
inventory(:brake,brake#1._);
inventory(:rim,rim#0._);
inventory(:rim,rim#1._);
inventory(:hub,hub#0._);
inventory(:hub,hub#1._);
inventory(:tire,tire#0._);
inventory(:tire,tire#1._);
inventory(:solidtire,solidtire#0._);
inventory(:solidtire,solidtire#1._);
inventory(:chain,chain._);
inventory(:belt,belt._);
inventory(:shifter,shifter#0._);
inventory(:shifter,shifter#1._);
inventory(:seatpost,seatpost._);
inventory(:saddle,saddle._);
inventory(:sprockets,sprockets._);
```

Prolog excerpt for `inventory/2` predicate. Note the row with the default values:

```
inventory(brake,[Type,Weight,Cost]):-
    member([Type,Weight,Cost],[[0,0,0], [1,25,50], [2,100,200]]).
```

APPENDIX B

COMPILER USAGE

B.1 Source Code

The source code repository is available at [13]. The list of source files and their related components (as shown in figure 4.1) are given in table B.1.

B.2 Build Instructions

For Windows, build command is: `make -f Makefile.mingw` . For Linux, running `make` is sufficient. An executable named `compiler` is generated. Tested build environment is GCC version 7, Make version 4.2, Lemon version 1.0, Ragel version 6.9.

B.3 Running Instructions

Feature model compiler takes an input file parameter, optionally an output file parameter and a template parameter. Output file parameter can be omitted in which case the output is printed to the console. Running `fmdsl` executable without any options displays a help message:

```
Usage: fmdsl [-t template][-p][-c4][-g5][-g6] input [output]
Options:
    -t    Template file with marker /**EMIT**/
          replaced with the output.
    -p    Output in GNU prolog. (default)
    -c4   Output in java for Choco 4.
    -g5   Output in C++ for Gecode 5.
    -g6   Output in C++ for Gecode 6.
```

Table B.1: Source Files

Files	Components
`compiler.c`	Main input & output, Options
`errors.h errors.c`	Error messages
`lexer.h lexer.rl`	Lexer
`parser.h parser.c grammar.lm`	Parser
`tree.h tree.c`	FM AST
`convert.h convert.c`	Tree Converter
`emit.h emit.c`	CLP IR
`gnu_prolog_emit.h`	GNU Prolog Translator
`gnu_prolog_emit.c`	
`gnu_prolog_support.pl`	GNU Prolog Utilities
`choco4_emit.h`	Choco 4 Translator
`choco4_emit.c`	
`choco4_support.java`	Choco 4 Utilities
`choco4_template.java`	
`gecode5_emit.h`	Gecode 5 Translator
`gecode5_emit.c`	
`gecode5_support.cpp`	Gecode 5 Utilities
`gecode6_emit.h`	Gecode 6 Translator
`gecode6_emit.c`	
`gecode6_support.cpp`	Gecode 6 Utilities
`Makefile Makefile.mingw`	Build files
`makestring._c`	
`tests/`	Test inputs

APPENDIX C

EXAMPLE ANALYSIS SESSION

Example given in section A.5 is the source file of a simplified model which describes a bike. Provided by the user, `inventory/2` predicate is for describing inventory for various features of the bike model and form the basis of example queries described here.

C.1 Partial Configurations, Counting and Filtering

To understand product configurations, output of `product_configuration/1` should be examined first:

```
?-  product_configuration(C).

C = [bike-[total_cost-_,total_weight-_],frame-[
material-_,hasdisc-_,weight-_,cost-_],wheel_0,wheel_1,
drivetrain,brake_0-[type-_,weight-_,cost-_],brake_1-[
type-_,weight-_,cost-_],seat,accessories,rim_0-[
material-_,diameter-_,holes-_,weight-_,cost-_],
rim_1-[material-_,diameter-_,holes-_,weight-_,cost-_],
spokes_0-[length-_,count-_],spokes_1-[length-_,count-_],
hub_0-[type-_,diameter-_,holes-_,hasdisc-_,gears-_,
spline-_,weight-_,cost-_],hub_1-[type-_,diameter-_,
holes-_,hasdisc-_,gears-_,spline-_,weight-_,cost-_],
tire_0-[diameter-_,weight-_,cost-_],tire_1-[diameter-_,
weight-_,cost-_],solidtire_0-[diameter-_,weight-_,
```

```
cost-_],solidtire_1-[diameter-_,weight-_,cost-_],
brakedisc_0,brakedisc_1,tube_0-[diameter-_],tube_1-[
diameter-_],chain-[weight-_,cost-_],belt-[weight-_,
cost-_],shifter_0-[gears-_,cost-_],shifter_1-[gears-_,
cost-_],crankset,sprockets-[count-_,cost-_],pulley,
tensioner,chainrings-[count-_],crankpulley,
derailleur_0-[type-_,gears-_],derailleur_1-[type-_,
gears-_],seatpost-[material-_,weight-_,cost-_],
saddle-[weight-_,cost-_]]
```

`C` is a list of feature names and their attributes list if it exists. It is similar to the output of `product_valid/3` but not cannot be used as a partial or complete configuration directly. Feature names need to be prefixed with + or − operators to construct a configuration. A valid configuration can be seen by examining an output of `product_valid/3`:

```
?- product_valid(V,[],true).

V = [bike-[total_cost-1060,total_weight-9960],frame-[
material-1,hasdisc-0,weight-7950,cost-100],+wheel_0,
+wheel_1,+drivetrain,brake_0-[type-1,weight-25,cost-50],
brake_1-[type-1,weight-25,cost-50],+seat,+accessories,
rim_0-[material-2,diameter-16,holes-36,weight-150,
cost-50],rim_1-[material-2,diameter-16,holes-36,
weight-150,cost-50],spokes_0-[length-6,count-36],
spokes_1-[length-6,count-36],hub_0-[type-1,diameter-5,
holes-36,hasdisc-0,gears-0,spline-0,weight-50,cost-30],
hub_1-[type-3,diameter-5,holes-36,hasdisc-0,gears-0,
spline-7,weight-100,cost-100],tire_0-[diameter-16,
weight-80,cost-40],tire_1-[diameter-16,weight-80,
cost-40],-solidtire_0,-solidtire_1,+brakedisc_0,
+brakedisc_1,tube_0-[diameter-16],tube_1-[diameter-16],
-chain,belt-[weight-50,cost-400],-shifter_0,-shifter_1,
```

```
+crankset,-sprockets,+pulley,+tensioner,-chainrings,
+crankpulley,-derailleur_0,-derailleur_1,seatpost-[
material-1,weight-300,cost-50],saddle-[weight-1000,
cost-100]] ?
```

Here we can see that our model is a valid model and the first valid configuration found. Next, a partial configuration is given where all material attributes are set to `3` and validity is confirmed:

```
?- product_valid([_,frame-[material-3,hasdisc-_,weight-_,
cost-_],_,_,_,_,_,_,_,rim_0-[material-3,_,_,_,_],rim_1-[
material-3|_],_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,
_,_,_,seatpost-[material-3|_],_],[],true).

true ?
```

A trivial example of invalid configuration is where the root feature is unselected:

```
?- product_valid([-bike|_],[],true).

no
```

An example of invalid partial configuration is given, where all materials are set to 3 and `solidtire_0` and `solidtire_1` are selected:

```
?- product_valid([_,frame-[material-3,hasdisc-_,weight-_,
cost-_],_,_,_,_,_,_,_,rim_0-[material-3,_,_,_,_],rim_1-[
material-3|_],_,_,_,_,_,_,solidtire_0-_,solidtire_1-_,_,
_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,seatpost-[material-3|_],_],
[],true).

no
```

Filtering is similar to checking a configuration but more efficient. `Filters` and `Goal` parameters of predicates described in section 5.1 is used for filtering. Previous example using filters;

```
?- product_valid(_,[+solidtire_0,+solidtire_1,
+frame-material=3,+rim_0-material=3,+rim_1-material=3,
+seatpost-material=3],true).

no
```

In this example all the materials are bound to variable `M` and value `3` is removed from possible values in the goal clause:

```
?- product_valid(_,[+frame-material=M,+rim_0-material=M,
+rim_1-material=M,+seatpost-material=M],M#\=3).

M = 2 ?
```

Counting configurations are similar to finding valid configurations. `product_count/3` works the same as `product_valid/3` but outputs a count of valid configurations instead. Counting all valid products:

```
?- product_count(C,[],true).

C = 306800
```

C.2 Analyzing Feature Occurrences

In section C.1 section the example which selected `solidtire_0` did not yield a valid configuration. Turns out it is a dead feature in the feature model, caused by expression `rim#0.diameter==tire#0.diameter` which implicitly selected `tire#0` always and excluded `solidtire#0` as a consequence. Similarly for `solidtire#1`;

```
?- features_dead(D,[],true).

D = [solidtire_0,solidtire_1]
```

Examining optional features where `shifter_1` is selected:

```
?- features_optional(O,[+shifter_1],true).

O = [accessories,brakedisc_0,brakedisc_1,derailleur_0]
```

Curiously `derailleur#0` is optional although `shifter#0` is not. A cursory look into the model might suggest a shifter needs a derailleur but this is not the case. Removing hub type `4` which is defined as hub gear and can have a shifter changes the optional features. Checking the core feature set confirms `derailleur#0` thus must be selected:

```
?- features_core(C,[+shifter_1,+hub_1-type=H],H#\=4).

C = [bike,frame,wheel_0,wheel_1,drivetrain,brake_0,
brake_1,seat,rim_0,rim_1,spokes_0,spokes_1,hub_0,hub_1,
tire_0,tire_1,tube_0,tube_1,chain,shifter_0,shifter_1,
crankset,sprockets,tensioner,chainrings,derailleur_0,
derailleur_1,seatpost,saddle];
```

C.3 Global Constraints and Optimization

More constraints on a valid configuration can be specified by arbitrary prolog clauses. The predicate to call and its parameters are declared in the model source. The predicate itself should receive integer or boolean values and nested lists of attribute or feature variables. The source can also pass an identifier in the host environment untouched. Some of the useful constraint examples are given here.

C.3.1 sum

Filter expressions in the form `attr=L` are convenient way to collect all attributes named `attr` of the selected features. The attributes of unselected values are set to `0` so it can be conveniently used for summation. `sum_eq/2` is defined to sum CLP variables;

```
?- product_valid(Z,[cost=_CL,weight=_WL],(sum_eq(C,_CL),
sum_eq(W,_WL))).

C = 1060
W = 9960
Z = [bike-[total_cost-1060,total_weight-9960],frame-[
material-1,hasdisc-0,weight-7950,cost-100],+wheel_0,
+wheel_1,+drivetrain,brake_0-[type-1,weight-25,cost-50],
brake_1-[type-1,weight-25,cost-50],+seat,+accessories,
rim_0-[material-2,diameter-16,holes-36,weight-150,
cost-50],rim_1-[material-2,diameter-16,holes-36,
weight-150,cost-50],spokes_0-[length-6,count-36],
spokes_1-[length-6,count-36],hub_0-[type-1,diameter-5,
holes-36,hasdisc-0,gears-0,spline-0,weight-50,cost-30],
hub_1-[type-3,diameter-5,holes-36,hasdisc-0,gears-0,
spline-7,weight-100,cost-100],tire_0-[diameter-16,
weight-80,cost-40],tire_1-[diameter-16,weight-80,
cost-40],-solidtire_0,-solidtire_1,+brakedisc_0,
+brakedisc_1,tube_0-[diameter-16],tube_1-[diameter-16],
-chain,belt-[weight-50,cost-400],-shifter_0,-shifter_1,
+crankset,-sprockets,+pulley,+tensioner,-chainrings,
+crankpulley,-derailleur_0,-derailleur_1,seatpost-[
material-1,weight-300,cost-50],saddle-[weight-1000,
cost-100]] ?
```

C is the total cost and W is the total weight of the configuration P. _CL is individual costs and _WL is individual weights. Domains in the model need not match so care must be taken to the interpretation of the results.

C.3.2 minimize, maximize

Using fd_maximize/2 and fd_minimize/2 useful configurations can be found. Finding cheapest bikes with the most gears and two shifters:

```
?- fd_maximize(fd_minimize(product_valid(P,[
+bike-total_cost=C,+shifter_0-gears=_G0,
+shifter_1-gears=_G1],_G0*_G1#=G),C),G).

C = 940
G = 21
P = [bike-[total_cost-940,total_weight-10500],frame-[
material-1,hasdisc-0,weight-7950,cost-100],+wheel_0,
+wheel_1,+drivetrain,brake_0-[type-1,weight-25,cost-50],
brake_1-[type-1,weight-25,cost-50],+seat,+accessories,
rim_0-[material-2,diameter-26,holes-36,weight-300,
cost-30],rim_1-[material-2,diameter-26,holes-36,
weight-300,cost-30],spokes_0-[length-11,count-36],
spokes_1-[length-11,count-36],hub_0-[type-1,diameter-5,
holes-36,hasdisc-0,gears-0,spline-0,weight-50,cost-30],
hub_1-[type-3,diameter-5,holes-36,hasdisc-0,gears-0,
spline-7,weight-100,cost-100],tire_0-[diameter-26,
weight-150,cost-20],tire_1-[diameter-26,weight-150,
cost-20],-solidtire_0,-solidtire_1,+brakedisc_0,
+brakedisc_1,tube_0-[diameter-26],tube_1-[diameter-26],
chain-[weight-150,cost-100],-belt,shifter_0-[gears-3,
cost-40],shifter_1-[gears-7,cost-70],+crankset,
sprockets-[count-7,cost-150],-pulley,+tensioner,
chainrings-[count-3],-crankpulley,derailleur_0-[
```

```
type-1,gears-3],derailleur_1-[type-2,gears-7],
seatpost-[material-1,weight-300,cost-50],saddle-[
weight-1000,cost-100]] ?
```

`G` is maximized among the minimum cost (`C`) bikes. Using variables in the `Goal` clause makes optimization operation more efficient than using it elsewhere.

C.3.3 all_different

In the model there are no restrictions for the materials of the rims, so they can be different. Eliminating those cases can reduce the valid configurations by nearly 2/5 so it may be better to include it in the model itself, Counting such undesirable cases;

```
?- product_count(C,[+rim_0-material=M0,
+rim_1-material=M1],all_different([M0,M1])).

C = 117600
```