Automated Reasoning on Feature Models via Constraint Programming

Carlos Eduardo Alvarez Divo
Abstract

Automated Reasoning on Feature Models via Constraint Programming

Carlos Eduardo Alvarez Divo

Feature models are often used in software product lines to represent a set of products and reason over their properties, similarities and differences, costs, etc. The problem becomes automating such reasoning which translates into a positive impact in terms of production, cost, and creation of the final products. To approach this matter we take advantage of the benefits of the constraint programming technology, which has proven to be most effective when solving problems of large complexity. Throughout the thesis we state the reasons for choosing this tool, evaluating its advantages and drawbacks, and showing results that support the conveniences of using constraint programming.

Keywords: feature models, software product lines, constraint programming.
## Contents

1 Introduction 3

2 Background 4  
   2.1 Software Product Lines .......................... 5  
   2.2 Feature Models .................................. 5  
   2.3 Constraint Programming ......................... 6  
      2.3.1 Definitions ................................ 6  
      2.3.2 How a CP Solver Works ..................... 10

3 A Description of the Problem 11  
   3.1 Overview ......................................... 11  
   3.2 Operations ....................................... 13  
   3.3 Reasons for Choosing CP .......................... 14

4 Related Studies 15  
   4.1 Overview ......................................... 15  
   4.2 Mapping an FM to a CSP ........................... 15  
   4.3 CP Solvers ....................................... 16

5 Methodology 16  
   5.1 Sample Models ..................................... 17  
   5.2 FM to CSP: A New Mapping ....................... 17  
   5.3 CSP to SICStus Prolog ............................ 18  
   5.4 Using Global Constraints .......................... 20  
   5.5 Using Branching Heuristics ....................... 20  
   5.6 Equivalence of FMs ................................ 21  
      5.6.1 Naïve Approach ............................. 21  
      5.6.2 Non-Naïve Approach ......................... 23  
      5.6.3 Logical Negation of CP Models ............... 24  
      5.6.4 Negation of SICStus Constraints ............ 24  
      5.6.5 New Algorithms, Inputs and Results ......... 25

6 The Solution 26  
   6.1 Overview ......................................... 26  
   6.2 The New Mapping .................................. 27  
      6.2.1 Motivation .................................. 27  
      6.2.2 Outcome .................................... 28
6.3 The Analysis Operations ........................................ 29
  6.3.1 Overview .................................................. 29
  6.3.2 The New Operations ...................................... 29
6.4 Best Branching Heuristic ..................................... 30

7 Critical Analysis of the Solution 31
  7.1 Highlights .................................................... 31
  7.2 Benefits of the New Mapping ................................. 32
  7.3 The Operations .............................................. 32
  7.4 Performance of the Operations ............................... 33

8 Critical Analysis of the Work 33
  8.1 Contributions ................................................ 33
  8.2 Difficulties ................................................... 35
  8.3 Future Work .................................................. 35

9 Comparison With Related Work 36
  9.1 First Proposals and Mapping ................................. 36
  9.2 Operations, Sample Feature Models, and CP Solvers .... 36
  9.3 A Final Word Regarding Improvements .................... 37
Acknowledgements

Numerous things took part in my year in Sweden as an exchange student and during the writing of my master thesis. I’ll take this opportunity to thank the people that made this possible.

I am grateful to my family, specially my mother, my father, and my brother who have given me their full support, and have kept me motivated doing what I do, not only during the period in which I wrote this thesis but ever since I started my undergraduate studies in 2006.

What I know about the constraint programming technology I learned from Professor Pierre Flener. This master thesis wouldn’t have been possible without his involvement, since he has been tutoring me every step of the way, and it was him who motivated me to take on the project in the first place. Mr Ahmet Serkan Karataş, and Professors Halit Oğuztüzün and Ali Doğru of the Middle East Technical University in Ankara, Turkey, have also taken an important part in the project since they cooperated with Prof. Flener and myself from the beginning and also provided both the feature models (Figures 1 and 2) and their SICStus Prolog implementation with which they worked in [1, 2] for us to study and try to improve. For that, I thank all of them.

I am also thankful to both my coordinators from Simón Bolívar University and Uppsala University, Professor Soraya Abad and Professor Roland Bol respectively. They’ve helped me with most of the administrative chores and have also given me advice for choices that I’ve had to make regarding my career, during my year in Sweden.

Finally, I’d like to thank Miss Noelia Ollvid and the rest of the people working at the International Office, and all the friends I’ve made since I came here that have made this a once-in-a-lifetime experience.

1 Introduction

According to the IEEE, a software feature is "a distinguishing characteristic of a software item" [7]. A feature model (FM) is a hierarchically arranged set of features, the relationships among these features that determine the composition rules and the cross-tree constraints, and some additional information, such as trade-offs, rationale, and justifications for feature selection [2].
FMs have been widely used in software product lines since their introduction by K. Kang et al. in 1990 [6], since they allow us to study the relations between products from the same SPL such as commonalities and variabilities [3, 2]. However, there are more complex analyses that can be made over FMs such as generating the set of all products derived from the FM, determining the size of such a set, deciding whether a particular product is valid, and deciding if two FMs are equivalent, among others.

Performing these operations is what we refer to as reasoning and for the past two decades automating this reasoning has been the main challenge towards FMs. The reason for this is that FMs are designed to promote product scalability, make it easier for features to be reusable, and to provide useful insights on product properties that can in turn reduce costs in production.

This work focuses on solving this matter using the global search methods of constraint logic programming over finite domains (CP from this point forward) which has been a very active area of research over the past decades and has shown to be very effective when solving large problems. We will show that this technology is well fit for the task because there is a very simple way to convert an FM into a constraint satisfaction problem (CSP). Moreover, once the CSP is formulated most of the translation into the CP solver (SICStus Prolog [5, 8] in this case) is very intuitive and allows the modeler to use extra tools such as the use of global constraints and the option to choose between different branching heuristics to try and improve the runtimes of the operations.

The remainder of this work is structured as follows. Section 2 presents the reader with some background information to establish the context of the problem. Section 3 gives a more detailed explanation of the problem. In Section 4 we explore other studies done in the same area, and we continue in Sections 5 and 6 pointing out the methods used to accomplish our solution, and a description of the solution itself. Section 7 contains an analysis on how well the problem was solved, and in Section 8.2 we discuss the difficulties we encountered. Finally, Section 8 presents a critical analysis on this work, and Section 9 presents a comparison with related studies introduced in Section 4.

2 Background

The following is a brief overview of some key concepts that will make the understanding of this work easier.
2.1 Software Product Lines

A software product line (SPL) or software production line is a software engineering methodology whose purpose is to create families of (often large and complex) systems instead of continuously developing individual products. SPLs focus on software reuse in an attempt to reduce costs and resources during production and development.

"Software product lines represent perhaps the most exciting paradigm shift in software development since the advent of high-level programming languages. Nowhere else in software engineering have we seen such breathtaking improvements in cost, quality, time to market, and developer productivity, often registering in the order-of-magnitude range" say F. J. van der Linden et al. in [9]. We refer the reader to Chapter 1 of this book to get a more comprehensive grasp of the SPL concept and motivation. And for further reading on this matter, Part II show experience reports on 10 different companies that adopted an SPL approach for the making of their products.

2.2 Feature Models

In [6] K. Kang et al. define features as the attributes of a system that directly affect end-users, and describe a feature model (FM) as a representation of the standard features of a family of systems in the domain and relationships between them. Several extensions for FM have been proposed. K. Czarnecki et al. introduced attributes, cardinalities for the decomposition relations, and reference attributes [10]. An attribute is a particular characteristic of a feature that is relevant to the FM and it is the only extension we will use for our FMs throughout this work. From this point on FMs refers to extended feature models with attributes unless stated otherwise.

The characteristics of an FM allow the modeler to describe the product family according to its key parts, namely the features, the relations between them (i.e., mandatory, optional, alternative, and or relations), their corresponding domain and the cross-tree relations (i.e., requires, excludes, and cross-tree constraints). Furthermore, they set the scope of the product family thus they can be used not only for documentation purposes but also for the system specification. In this work we consider explicit cross-tree constraints (e.g., “feature 3D Car Race Application requires Memory.size ≥ 512”) to be a part of the FM.

We will use the same notation for FMs that was used in [1, 2]. Such
notation is explained in Table 1 which we include for the sake of completeness.

Figures 1 and 2 are examples of FMs of a cellphone and a computer product family respectively.

2.3 Constraint Programming

2.3.1 Definitions

Constraint programming can be defined as a set of techniques such as algorithms or heuristics that deal with CSPs [11]. We refer the reader to [19] for a survey on the CP history. Throughout this thesis we use a terminology specific for the area of constraint programming. We now present some of the most important definitions.

A decision variable or simply variable, is a name that holds information regarding a value that is initially unknown.

The domain of a decision variable is the set of possible values that the variable can take.

Constraints are rules used to define a specific problem. As the name suggests they constrain the domain of the variables in a CSP to have only the values that satisfy them.

The reification of a constraint \( c \) is a new constraint \( c \Leftrightarrow b \) where \( b \) is a boolean value that becomes true if and only if the constraint is satisfied. Reification is a very useful tool because it enables us to model rules in the form “at least two of these constraints must be satisfied, the sum of these values must be between 2 and 4 inclusive, the number of peripherals can be 5 at most…”.

CSP stands for constraint satisfaction problem. A CSP is a triple \( (V, D, C) \) where \( V = \{v_1, v_2, \ldots, v_n\} \) is the set of variables of the problem, \( D = \{d_1, d_2, \ldots, d_n\} \) is the set of domains of every variable in the problem such that \( \forall i : v_i \in d_i \), and \( C = \{c_1, c_2, \ldots, c_m\} \) is the set of constraints.

\[
P = (\{x, y, z\}, \{\{1, 2, 3, 4\}, \{1, 2, 3, 4\}, \{1, 2, 3, 4\}\}, \{x > y, x + y < z\}) \quad (1)
\]

is an example of a CSP.

A store is a set of mappings from the variables to a subset of their own domain, i.e. \( \{v_i \mapsto d'_i \mid v_i \in V \land d_i \in D \land d'_i \subseteq d_i\} \). For instance (notice the notation):

\[
\{x \mapsto \{1, 2\}, y \mapsto \{1, 3, 4\}, z \mapsto \{3, 4\}\}
\quad (2)
\]
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="MandatoryRelation.png" alt="Diagram" /></td>
<td>Mandatory Relation</td>
<td>Let $P$ and $C$ be two features, where $P$ is the mandatory parent of $C$, in a Mandatory relation. Then, $C$ is included in a configuration if and only if $P$ is included in the configuration.</td>
</tr>
<tr>
<td><img src="OptionalRelation.png" alt="Diagram" /></td>
<td>Optional Relation</td>
<td>Let $P$ and $C$ be two features, where $P$ is the optional parent of $C$, in an Optional relation. If $P$ is included in a configuration then $C$ may or may not be included in the configuration.</td>
</tr>
<tr>
<td><img src="AlternativeRelation.png" alt="Diagram" /></td>
<td>Alternative Relation</td>
<td>Let $P, C_1, C_2, \ldots, C_n$ be features, where $P$ is the parent of $C_1, C_2, \ldots, C_n$, in an Alternative relation. If $P$ is included in a configuration, exactly one of the child features $C_1, C_2, \ldots, C_n$ must be included in the configuration.</td>
</tr>
<tr>
<td><img src="OrRelation.png" alt="Diagram" /></td>
<td>Or Relation</td>
<td>Let $P, C_1, C_2, \ldots, C_n$ be features, where $P$ is the parent of $C_1, C_2, \ldots, C_n$, in an Or relation. If $P$ is included in a configuration, a nonempty subset of $C_1, C_2, \ldots, C_n$ must be included in the configuration.</td>
</tr>
<tr>
<td><img src="FeatureAttribute.png" alt="Diagram" /></td>
<td>Feature Attribute ($F.attr$)</td>
<td>Feature attributes are used to supply some additional information (e.g. priority, cost, size, speed) about features. A feature attribute consists of a name, a domain, and a value (or subset) of its domain.</td>
</tr>
<tr>
<td><img src="Requires.png" alt="Diagram" /></td>
<td>Requires</td>
<td>If a feature $X$ requires a feature $Y$, the inclusion of $X$ in a configuration implies the inclusion of $Y$ in such a configuration.</td>
</tr>
<tr>
<td><img src="Excludes.png" alt="Diagram" /></td>
<td>Excludes</td>
<td>If a feature $X$ excludes a feature $Y$, the inclusion of $X$ in a configuration implies the exclusion of $Y$ in such a configuration.</td>
</tr>
</tbody>
</table>

Table 1: Notation for FMs.Courtesy of the authors of [1, 2]
Constraints

- Cross-tree: When there are two CPUs on board, Task Scheduler must be a part of the product (CPU 1 and CPU 2 implies Task Scheduler)

- Cross-tree: If Video Call.mpc ≥ 4 then Video Call requires Screen.resolution ≥ 320x640 and 3G Connector.speed ≥ 6

- Cross-tree: 3D Car Race requires (GPU and RAM.size ≥ 512) or (RAM.size ≥ 1024)

- Cross-tree: If Screen.resolution < 320x640 then Screen excludes GPS

- Cross-tree: Task Scheduler requires CPU 1.speed ≥ CPU 2.speed

**Figure 1:** FM of a cellphone product family. Courtesy of the authors of [1, 2]
- Global: Each PCI Card that is a part of the product must be installed on a different PCI slot.

- Global: Each data collector, 1 through 4, must be assigned to a data communication channel, 1 through 3. Each data communication channel has a designated capacity and these capacities must not be exceeded.

- Global: Total power consumption of the hardware parts cannot exceed the capacity of the power supply.

- Global: Total cost of a product cannot exceed a designated budget.

- Global: Memory size must be greater than the total memory consumption of the system software plus memory consumption of the application with the highest memory requirement among the applications chosen to be a part of the product.

- Cross-Tree: Task Scheduler requires $\text{CPU 1.speed} \geq \text{CPU 2.speed}$.

- Cross-Tree: Data Collectors 1 and 2 cannot be assigned to the same channel.

**Figure 2:** FM of a computer product family. Courtesy of the authors of [1, 2]
**Store strength.** Let us denote the domain of a variable \( x \) in a store \( s \) as \( \text{domain}_s(x) \). A store \( s_1 \) is (strictly) stronger than store \( s_2 \), and we denote it \( s_1 \prec s_2 \) iff for every decision variable \( x \) we have \( \text{domain}_{s_1}(x) \subseteq \text{domain}_{s_2}(x) \), and \( \text{domain}_{s_1}(x) \subset \text{domain}_{s_2}(x) \) for at least one decision variable.

A **propagator** is a program that enforces a constraint on some variables. For instance in the example above (1) let us run the propagator of the constraint \( x + y < z \) with domain consistency. The resulting store would be

\[
\{ x = \{1, 2\}, y = \{1, 2\}, z = \{3, 4\}\}
\]

Domain consistency means that for every value for every variable involved in a constraint, there is at least one valid value for every other variable in the constraint. For further reading on consistency we refer the reader to [13].

Now, let us introduce the following notation: \( p_c \) refers to the propagator of a constraint \( c \) and \( p(s) \) means running a propagator \( p \) on a store \( s \). Every propagator must be:

1. **Contracting:** running the propagator \( p_c \) of a constraint \( c \) on a store \( s \) results in a stronger or equal store, i.e. \( p_c(s) \preceq s \).

2. **Monotone:** strength-ordered stores remain ordered, i.e.
   \( s_1 \preceq s_2 \Rightarrow p(s_1) \preceq p(s_2) \).

3. **Able to identify a solution:** for a solution \( s \) to a constraint \( c \), no domain is shrunk, i.e. \( p_c(s) = s \).

An **assignment** is a store \( s \) where \( \forall i : |\text{domain}_s(x_i)| = 1 \). The following is an assignment for the example CSP given in (1) (same notation used with stores):

\[
\{ x \mapsto \{2\}, y \mapsto \{1\}, z \mapsto \{3\}\}
\]

A solution or **solution store** of a problem is an assignment for which all the constraints are satisfied. For example (3) is a solution of (1).

**Propagation** and search are discussed in the following subsection.

### 2.3.2 A General Idea About How a CP Solver Works

A CP solver is a software package that takes a problem modelled as a CSP and determines whether there exists a solution for the problem [11]. In general, solvers provide a framework to define the variables of the problem,
define their domains, post the constraints, and set the configuration of the search engine.

Once this is coded into the solver and the program starts running the two most important things that happen behind the scenes are *propagation* and *search*.

The properties of propagators make the process of solving a problem remain independent from the order they are executed. The selection of this individual propagators is done by a *master algorithm*, which keeps track of the propagator next in line to be executed, the subsumed propagators (i.e. the ones that can no longer change the current store), the moment from which search is needed, among other things.

Search is the process of selecting one variable and assigning to it a value of its domain. It is also referred to as branching and is used when propagation reaches a fixpoint. The branching is achieved by a propagator whose behaviour depends on the branching heuristics. For instance let us assume that propagation reached a fixpoint when running on a CSP and the current store is (2). Performing search on the *first unassigned variable* assigning to it its *minimum value* would mean posting the constraint \( x = 1 \) and running its corresponding propagator. After that (if possible) the process of propagation would start again, interleaving with search until a solution is found. There are means of knowing if propagation is at fixpoint. Also there are properties of the propagators and the master algorithm that improve the efficiency of propagation but they escape the scope of this work. Branching heuristics are further discussed in Section 5.5.

The solver we worked with in this study is SICStus Prolog and the Constraint Logic Programming over Finite Domain library [5, 8] that it provides. Such library is capable of handling CSPs with boolean and integer variables, arithmetic, logical, and global constraints, variable and value orderings, reification, etc.

# 3 A Description of the Problem

## 3.1 Overview

Ever since the Industrial Revolution, everything has been produced massively and automatically with the support of machines and computers. Its most important consequence is what every company aims for: decrease in
production costs and therefore increase in revenues.

Nowadays the efforts concentrate on improving every process involved in the massive production of goods, from designing them to developing them to delivering them to the customers and everything in between. Engineering methodologies have grown around this matter which is where SPLs come into the picture.

The idea behind SPLs is to focus on a family of products or systems instead of focusing on a single one. The number of products within a family of products might be very large and even though two products are different from each other they share several common characteristics. These commonalities describe the family of products.

SPLs aim to reduce the cost of production, promoting the reuse of components and in consequence increasing the developer productivity, making it easier to modify products in order to adapt them to new markets or integrate them with other products [9].

A way of modeling product families is with the use of feature models (FM), first introduced in 1990 by K. Kang et al. [6]. Since then their popularity has grown immensely and they are used as a means of specification and documentation. FMs constitute an elegant way of representing product families. FMs can often represent hundreds of thousands of products which is why automating the analyses on them is the only feasible alternative.

Let us consider computers for example. What are the parts of most of today’s modern computers? We can first think of the two most general components: hardware and software. Hardware including one or more processors, a hard drive, a RAM memory, I/O devices such as a screen, a keyboard, and a mouse. Software would imply for instance an OS, web browser, games, and an e-mail client. These could be the common things that all computers of a specific family share in common, but they could differ in many aspects such as whether or not they have a GPU or a graphics card, their size, their price, among other things. It is not a coincidence that we chose this particular example. In this study one of the FMs we work with is one representing a set of computers (Figure 2).

The next step is answering questions that may be interesting to the stakeholders of the product, such as

- How many products can we derive from the FM?
- What is the cheapest/most expensive product?
What features are included in all of the products?

To answer these questions a series of operations have been defined. In the following subsection we discuss such operations.

### 3.2 Operations

The analysis on FM consists of operations whose purpose is to answer questions that may be of use to the stakeholders of a product. Most of the operations of interest to this work are described in [2, 3, 4]. We shall briefly explain them for the sake of completeness as we introduce a few more, namely core attributes, variant attributes, dead attributes, and attributes description, but first we will present a couple of concepts used in the description of some of the operations.

Given an FM $M$, a total configuration or product of $M$ is a configuration in which all of the values of the features’ attributes have been set, as opposed to a partial configuration in which there is at least one attribute whose value hasn’t been set.

The following are the analysis operations for a given FM $M$:

1. **Void** Does $M$ represent any products?
2. **Valid product** Does a given product belong to the set of products represented by $M$?
3. **Valid partial configuration** Is a given partial configuration valid (i.e. does it not include any contradiction) with respect to $M$?
4. **All products** Compute all products represented by $M$.
5. **Number of products** Compute the number of products represented by $M$.
6. **Commonality** Compute the percentage of products represented by $M$ including a given configuration.
7. **Filter** Compute all products, including a given configuration, represented by $M$.
8. **Core features** Compute the set of features that are part of all the products.
9. **Variant features** Compute the set of features that appear in some but not all of the products.

10. **Dead features** Compute the set of features that are not part of any product.

11. **False optional features** Compute the set of features that, although modeled as optional, are part of all the products.

12. **Optimization** Compute the products fulfilling the criteria established by the given objective function.

13. **Core attributes** Compute the set of values that are part of all the products.

14. **Variant attributes** Compute the set of values that appear in some but not all of the products.

15. **Dead attributes** Compute the set of values that are not part of any product.

16. **Attributes description** Compute the set of core, variant, and dead attributes of $M$.

17. **Equivalence** Given two FMs $M_1$ and $M_2$, are the sets of products of $M_1$ and $M_2$ equal?

Other operations have been defined, for instance *multi-step configuration* and *corrective explanations* in [15], and [16] respectively, however this work is focused only in the operations listed above.

### 3.3 Reasons for Choosing CP

Over the years many approaches other than using CP have been proposed for reasoning on FMs such as converting the FM into propositional logic and using a SAT or BDD solver, translating it into description logic and using description logic reasoners, building ad-hoc data structures and algorithms for the FM, among others [11, 3]. In this section we explain the main reasons why we chose CP as we point out some of its advantages.
Converting an FM into a CSP and then coding it into a solver is very straightforward.

A lot of analysis operations can be implemented.

Constraint solvers provide excellent readability of the programs even to the unfamiliar eye.

CP has proven to work very well on large real life problems.

Not many performance analyses on how the different tools behave in practice have been published.

All solvers have built-in benchmarking functions to obtain information about runtimes, number of backtracks, size of the search space, etc.

It is possible to make experimental trials, for instance using different variable and value orderings, levels of consistency, global constraints, etc.

4 Related Studies

4.1 Overview

There are different approaches to automate the reasoning on FMs, namely: translating FMs into propositional logic, using ad-hoc data structures and algorithms for FMs, representing FMs with description logic, and using CP [11]. In this work we concentrate on the studies that have presented a way to translate FMs to CSPs, the ones that have used a CP solver for the analysis, and the ones that have published experimental results.

We refer to [11] for an extensive review on the rest.

4.2 Mapping an FM to a CSP

The first to propose using CP to reason on FM were Benavides et al. in [12, 14] where they provide a way to map FMs to CSPs. The key to this mapping is that the features make up the set of variables and the attributes are modeled as constraints.

Later, A. S. Karataş et al. proposed a new mapping from extended feature models, which may include complex feature-feature, feature-attribute, and
attribute-attribute cross-tree relationships, to constraint logic programming over finite domains [1]. Contrary to the previous mapping, in this one the attributes make up the set of variables. Both mappings model the relations of the FM in the same way.

In this work we will be using a slightly modified version of the mapping proposed in [1, 2], discussed in Section 5.2.

4.3 CP Solvers

As of 2009 the following CP solvers were used in research work to study reasoning on FMs: JaCoP, Choco, OPL, and GNU Prolog [11]. However only a limited amount of studies presented experimental results and only a handful of analysis operations were implemented. Table 2 shows a list of studies up until 2009 along with their corresponding operations.

<table>
<thead>
<tr>
<th>Study</th>
<th>CP Solver</th>
<th>Operations</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>J. White et al.</td>
<td>Choco</td>
<td>Valid Product, Corrective Explanations</td>
<td>[16]</td>
</tr>
<tr>
<td>D. Benavides et al.</td>
<td>OPL</td>
<td>All products</td>
<td>[4, 12, 14]</td>
</tr>
<tr>
<td>D. Benavides et al.</td>
<td>JaCoP, Choco</td>
<td>Void FM, Number of products</td>
<td>[17]</td>
</tr>
</tbody>
</table>

Table 2: Studies that used CP solvers and presented experimental results up until 2009 [11].

In 2010 A. S. Karataş et al. implemented 12 analysis operations (the first 12 operations defined in Section 3.2) using the SICStus Prolog solver and published their results [2]. In this work we present results for all 17 analysis operations, which were tested using the same solver.

5 Methodology

As we have mentioned before, using CP is a good approach for reasoning on FMs. In this section we explain the steps we followed and the decisions we made that lead to what we consider is a very good outcome.
5.1 Sample Models

The analysis operations we mentioned in Section 3 were tried on three different feature models. The first two FMs and their corresponding SICStus Prolog implementations (used in [1, 2]), were provided by Ahmet Serkan Karataş, Halit Oğuztüzün, and Ali Doğru: a small FM representing a family of cellphones with 256 possible products shown in Figure 1, and a medium size FM of a family of computers shown in Figure 2, with a total of 338,928 products. With these two FMs we built a bigger one to verify if the approach scaled well on larger FMs; we call this the package model. As shown in Figure 3 a package is a compound of one computer and one cellphone. We derive a total of 86,765,568 (= 338,928 · 256) packages from this FM.

5.2 FM to CSP: A New Mapping

With the FMs at hand, the next step is to transform them into a CSP. In this study we propose a modification of the mapping described [1] in an attempt to reduce the work of both the modeler and the solver, and perhaps improving its performance.
We now present the difference between our mapping with respect to the one in [1]. The computer FM is shown in Figure 2.

**Definition.** Every feature has an implicit attribute named *selected* that ranges over the domain \{true, false\}. This attribute determines the presence or absence of a feature in a configuration. The name of a feature in an FM is the name we use for the boolean decision variable of the implicit attribute selected of such feature in the CP model.

**Definition.** A mandatory feature is a feature that can be reached from the root of the FM through a path of only mandatory relations, i.e., a feature that will be present in all of the products (the root of an FM is of course a mandatory feature). For instance the feature *Network* is a mandatory feature of the computer FM.

**Definition.** An optional feature is a feature that is not mandatory. The feature *CPU2* is an example of an optional feature in the computer FM.

In [1] the implicit attribute *selected* of every feature becomes a boolean variable of the CSP, while in our mapping, the implicit attribute *selected* of every optional feature in the FM becomes a boolean variable (i.e., with domain \{false, true\} or \{0, 1\}), whereas the implicit attribute *selected* of every mandatory feature becomes an assigned variable with domain \{true\} (or \{1\}).

The reasons for using this mapping are discussed in Section 6.

### 5.3 CSP to SICStus Prolog

Once the CSP is formulated coding it into SICStus Prolog is very straightforward, though it can be tedious in some cases when the FM is very large. We will come back to this issue in Section 8.3.

In order to program the FM one must do three things:

1. **The Constraint Logic Programming over Finite Domains** library must be included at the top of the file with the instruction
   ```prolog
   :-use_module(library(clpfd)).
   ```

2. **Post the constraints** of the CSP associated with the FM

3. **Set the branching heuristic** for the search engine to use once propagation can no longer do work and search is needed.

Table 3 shows examples on how to translate an FM into SICStus Prolog.
### Constraint SICStus Prolog Constraint

#### Membership Constraints
- \( Computer, Hardware \in \{0, 1\} \equiv \text{domain([Computer, Hardware],0,1)} \)
- \( CPU1\_speed \in \{1600, 1800, 2000\} \equiv \text{CPU1\_speed in \{1600, 1800, 2000\}} \)
- \( Scr\_res \in \{1, \ldots, 4\} \equiv \text{Scr\_res in 1..4} \)

#### Mandatory Relations
- \( Computer \) is the mandatory parent of \( Hardware \)
- \( Optional \) Relation
- \( DVD \) is the optional parent of \( Hardware \)
- \( Or Relation \)
- \( Games \) is the parent of \( Chess, TDCR, \) and \( Tetris \)

#### Alternative Relation
- \( PS \) is the parent of \( PS1, PS2, PS3 \) \( \equiv PS \Rightarrow (PS1 + PS2 + PS3 = 1) \)
  \((PS1, PS2, PS3 \) are boolean variables representing the impl. attribute selected) \)

#### Requires
- \( MacOS \) requires \( RAM\_size \geq 2048 \equiv MacOS \Rightarrow (RAM\_size \geq 2048) \)

#### Excludes
- \( PS2 \) excludes \( DVD \) \( \equiv \#\ (PS2 \#/\ DVD) \)

#### Global Constraints
- Each PCI Card should be \( \equiv \text{all\_different([Graphics\_sn, Sound\_sn, Network\_sn, FireW\_sn, WiFi\_sn])} \)

#### Reification
- \( CPU1\_speed > CPU2\_speed \text{ iff } b \equiv (CPU1\_speed \Rightarrow CPU2\_speed) \iff b \)

#### Branching
- Leftmost variable, first value \( \equiv \text{labeling([],[V1, V2, \ldots, Vn])} \)
  \((V1, V2, \ldots, Vn \) are the variables of the CSP) \)

---

**Table 3:** Examples of constraints and branching in SICStus Prolog
5.4 Using Global Constraints

“A global constraint is customarily used to describe a complex (that is, complicated, in an informal sense) constraint, with the number of variables often being a parameter. An appropriate constraint propagation for a global constraint is then taken care of by means of a special purpose algorithm. Modeling by means of global constraints is therefore often more efficient than relying on the general purpose constraint propagation algorithms” [13].

There are several advantages of using global constraints when modelling a CSP.

- More compact and intuitive CP models, because more expressive constraints are available.
- More efficient propagation, since we have more global information.
- Stronger consistency: consider one n-ary domain consistent all\_different (also called all\_distinct or distinct) constraint compared to $O(n^2)$ binary domain consistent constraints.

Examples of the improvement caused by the use of global constraints are presented in [2] with gains between 20% and 62%.

5.5 Using Branching Heuristics

As we mention before going from an FM to a CSP, and then to a CP program is rather simple and the solver can do a good job performing the operations even if we don’t give options to the search engine. However, most solvers include several methods to select variables and values during search. These variable and value orderings are what we refer to as branching heuristics and they can influence the overall performance of an operation.

These are the built in variable and value orderings of SICStus Prolog [5]

**Variable orderings:** the following options control the order in which the next variable is selected for assignment.

- **leftmost** The leftmost variable is selected. This is the default.
- **min** The leftmost variable with the smallest lower bound is selected.
- **max** The leftmost variable with the greatest upper bound is selected.
ff The first-fail principle is used: the leftmost variable with the smallest domain is selected.

ffc The most constrained heuristic is used: a variable with the smallest domain is selected, breaking ties by (a) selecting the variable that has the most constraints suspended on it, and (b) selecting the leftmost one.

Value orderings: the following options control the way in which choices are made for the selected variable X:

step Makes a binary choice between $X = B$ and $X \neq B$, where B is the lower or upper bound of X. This is the default.

enum Makes a multiple choice for $X$ corresponding to the values in its domain.

bisect Makes a binary choice between $X \leq M$ and $X > M$, where $M$ is the midpoint of the domain of $X$. This strategy is also known as domain splitting.

Further discussion on this is presented in Section 6.

5.6 Equivalence of FMs

5.6.1 Naïve Approach

Checking if two FMs are equivalent means comparing the sets of solutions of both FMs for equality. To the best of our knowledge no algorithm for the equivalence operation has been published. A naïve approach to solving this problem is represented in Algorithm 1.

It is not possible to assume that the solutions will come in the same order in both CSPs, hence an ordered one-to-one solution comparison cannot be made. However, since the solutions are all different if we ascertain that $|S_1| = |S_2|$, checking that every $s_2$ that belongs to $S_2$, belongs also to $S_1$ is not necessary. Thus we can improve the algorithm as shown in Algorithm 2.

Verifying $S_1 = S_2$ has a complexity of $O(n^2)$ where $n = |S_1| = |S_2|$. Running Algorithm 2 took 7752.90 seconds to run on equivalent computer CP models.
Let $C_1 = (V_1, D_1, C_1)$ and $C_2 = (V_2, D_2, C_2)$ be two CSPs corresponding to FMs $M_1$ and $M_2$ respectively (assuming that every variable in $V_1$ corresponds to a variable in $V_2$ and vice versa).

1: Generate the sets of solutions $S_1$ and $S_2$ of $C_1$ and $C_2$ respectively
2: for all $s_1 \in S_1$ do
3: if $s_1 \notin S_2$ then
4: return $C_1$ and $C_2$ are not equivalent
5: for all $s_2 \in S_2$ do
6: if $s_2 \notin S_1$ then
7: return $C_1$ and $C_2$ are not equivalent
8: return $C_1$ and $C_2$ are equivalent

Algorithm 1: Representation of the naïve approach

Let $C_1 = (V_1, D_1, C_1)$ and $C_2 = (V_2, D_2, C_2)$ be two CSPs corresponding to FMs $M_1$ and $M_2$ respectively (assuming that every variable in $V_1$ corresponds to a variable in $V_2$ and vice versa).

1: Generate the sets of solutions $S_1$ and $S_2$ of $C_1$ and $C_2$ respectively
2: if $|S_1| \neq |S_2|$ then
3: return $C_1$ and $C_2$ are not equivalent
4: for all $s_1 \in S_1$ do
5: if $s_1 \notin S_2$ then
6: return $C_1$ and $C_2$ are not equivalent
7: return $C_1$ and $C_2$ are equivalent

Algorithm 2: Our first implementation of the naïve approach

A second improvement was made taking advantage of the facts that the solutions are all different and that we check beforehand that $|S_1| = |S_2|$. Now, if the verification $s_1 \in S_2$ succeeds then we take away $s_1$ from the set $S_2$, reducing the number of elements in every step of the loop as shown in Algorithm 3.

Now verifying $|S_1| = |S_2|$ has a complexity of $O(n)$ in the best case and $O(n^2)$ in any other case. Running Algorithm 3 on equivalent computer CP models took 6,925.98 seconds (for the worst case), which is a 10.66% improvement over Algorithm 2. Table 4 shows runtimes of several tests of Algorithms 2 and 3.

In any case, the naïve approach has a downside, because the solutions
Let $C_1 = (V_1, D_1, C_1)$ and $C_2 = (V_2, D_2, C_2)$ be two CSPs corresponding to FMs $M_1$ and $M_2$ respectively (assuming that every variable in $V_1$ corresponds to a variable in $V_2$ and vice versa).

1: Generate the sets of solutions $S_1$ and $S_2$ of $C_1$ and $C_2$ respectively
2: if $|S_1| \neq |S_2|$ then
3: \hspace{1em} return $C_1$ and $C_2$ are not equivalent
4: for all $s_1 \in S_1$ do
5: \hspace{1em} if $s_1 \in S_2$ then
6: \hspace{2em} $S_2 \leftarrow S_2 \setminus \{s_1\}$
7: \hspace{1em} else
8: \hspace{2em} return $C_1$ and $C_2$ are not equivalent

\textbf{Algorithm 3:} Improvement of Algorithm 2

have to be stored in lists. When we tried to test the algorithms mentioned above on the larger FMs we ran out of memory. We will revisit this issue in Section 8.2.

5.6.2 Non-Naïve Approach

In our attempt to implement an equivalence operation that works with very large models we decided to explore another alternative, using some basic set theory as follows:

Let $S_1$ and $S_2$ be two sets, then:
\begin{align*}
S_1 &= S_2 \\
\iff & S_1 \subseteq S_2 \land S_2 \subseteq S_1 \\
\iff & \forall s : (s \in S_1 \Rightarrow s \in S_2) \land (s \in S_2 \Rightarrow s \in S_1) \\
\iff & \forall s : (s \notin S_1 \lor s \in S_2) \land (s \notin S_2 \lor s \in S_1) \\
\iff & \forall s : -(s \in S_1 \land s \notin S_2) \land -(s \in S_2 \land s \notin S_1)
\end{align*}

Therefore, we can prove equivalence between to CSPs $C_1$ and $C_2$ if we can prove that the sets of solutions of $C_1 \land \neg C_2$ and $C_2 \land \neg C_1$ are both empty.
5.6.3 Logical Negation of CP Models

In order to implement the equivalence operation we need to provide a way to post the logical negation of a CP model which in turn is, the logical negation of its corresponding CSP. As we have mentioned before a CSP is a triple \((V, D, C)\) where \(V = \{v_1, v_2, \ldots, v_n\}\) is the set of variables, \(D = \{d_1, d_2, \ldots, d_n\}\) is the set of domains of every \(v_i\) such that \(\forall i : v_i \in d_i\); and \(C\) is the set of constraints of the problem.

To obtain the negation of \((V, D, C)\), for every constraint \(c_i \in C\) we post the reification of the negation of \(c_i\), i.e. \(\neg c_i \iff b_i\), and add the new constraint enforcing the sum of all the \(b_i\) to be strictly positive.

Formally speaking, given a CSP \(P = (V, D, C)\), where \(C = \{c_1, c_2, \ldots, c_n\}\) we define the negation of \(P\) as \(\neg P = (V \cup B, D \cup \bigcup_{i=1}^{n} \{0, 1\}, C' \cup C'')\) where

- \(C' = \{c'_1, c'_2, \ldots, c'_n\}\),
- \(c'_i = \neg c_i \iff b_i\), and
- \(B = \{b_1, b_2, \ldots, b_n\}\)
- \(C'' = \{\sum_{i=1}^{n} b_i > 0\}\).

5.6.4 Negation of SICStus Constraints

To obtain the negation of a CP model we need a way to negate and reify the constraints. This is very straightforward for simple (not global) constraints in SICStus Prolog, since we can enclose the constraint in parenthesis, put a logical-not operator in front of it, and make it logically equivalent to a boolean decision variable. For instance the reification of the negation of

\[
\text{(Sound } \# / \backslash \text{Graphics) } \# \Rightarrow (\text{Graphics}_\text{sn} \ #< \text{Sound}_\text{sn}) \]

is

\[
\# \backslash (\text{(Sound } \# / \backslash \text{Graphics) } \# \Rightarrow (\text{Graphics}_\text{sn} \ #< \text{Sound}_\text{sn})) \# \Rightarrow \text{B.}
\]

Unfortunately SICStus prolog (v4.1.3) doesn’t have built-in reifiable versions of the global constraints (this issue is discussed further in Section 8.2). In order to reify them we had to decompose them into several simple constraints, and then reify each one of them. For instance the following \textit{all different} constraint of the computer CP model
all_different([Graphics_sn, Sound_sn, Network_sn, FireW_sn, WiFi_sn])

was translated into the negation of the model as follows:

#\((\text{Graphics}\_\text{sn} \neq \text{Sound}\_\text{sn}) \iff B1\),
#\((\text{Graphics}\_\text{sn} \neq \text{Network}\_\text{sn}) \iff B2\),
#\((\text{Graphics}\_\text{sn} \neq \text{FireW}\_\text{sn}) \iff B3\),
#\((\text{Graphics}\_\text{sn} \neq \text{WiFi}\_\text{sn}) \iff B4\),
#\((\text{Sound}\_\text{sn} \neq \text{Network}\_\text{sn}) \iff B5\),
#\((\text{Sound}\_\text{sn} \neq \text{FireW}\_\text{sn}) \iff B6\),
#\((\text{Sound}\_\text{sn} \neq \text{WiFi}\_\text{sn}) \iff B7\),
#\((\text{Network}\_\text{sn} \neq \text{FireW}\_\text{sn}) \iff B8\),
#\((\text{Network}\_\text{sn} \neq \text{WiFi}\_\text{sn}) \iff B9\),
#\((\text{FireW}\_\text{sn} \neq \text{WiFi}\_\text{sn}) \iff B10\)

The decomposition of the three global constraints we used, namely sum, global cardinality, and all different, was taken from an implementation without global constraints of the computer FM provided by the authors of [1, 2].

5.6.5 New Algorithms, Inputs and Results

With the computer and the package CP models we built two more, replacing in each of them the constraint

\(\text{Cost} \#=\text{Budget}\)

with the constraints

\(\text{Cost} \#< 1300 \iff \text{Cost} = \text{Budget},\) and
\(\text{Cost} \#>= 1300 \iff \text{Cost} = \text{Budget} + 1\)

in order to alter the sets of solutions without affecting their cardinality. We call these models computer' and package'.

Algorithms 4 and 5 show our implementations of the equivalence operation using the non-naïve approach.

The following tests were run using the different algorithms of the equivalence operation as shown in Table 4, where \(\text{Equivalence}(C_1, C_2)\) denotes the
process of running an implementation of the equivalence operation on the CP models \( C_1 \) and \( C_2 \): 

**Test 1** Equivalence\((computer, computer)\)

**Test 2** Equivalence\((computer, computer')\)

**Test 3** Equivalence\((package, package)\)

**Test 4** Equivalence\((package, package')\)

Tests 1 and 3 are meant to be equivalence tests, and tests 2 and 4 are non-equivalence tests.

Let \( C_1 = (V_1, D_1, K_1) \) and \( C_2 = (V_2, D_2, K_2) \) be two CSPs corresponding to FMs \( M_1 \) and \( M_2 \) respectively. Assuming that every variable in \( V_1 \) corresponds to a variable in \( V_2 \) and vice versa 

1. while there are still solutions of \( C_1 \) to be found do
2. \( s_1 \leftarrow \) next solution of \( C_1 \)
3. if \( s_1 \) is a solution of \( \neg C_2 \) (no contradiction is found) then
4. return \( C_1 \) and \( C_2 \) are not equivalent
5. while there are still solutions of \( C_2 \) to be found do
6. \( s'_1 \leftarrow \) next solution of \( C_2 \)
7. if \( s'_1 \) is a solution of \( \neg C_1 \) (no contradiction is found) then
8. return \( C_1 \) and \( C_2 \) are not equivalent
9. return \( C_1 \) and \( C_2 \) are equivalent

**Algorithm 4:** First implementation of the non-naïve approach

6 The Solution

6.1 Overview

The aim of this thesis is to present a solution to the problem of automating the reasoning on FMs. In the previous section we presented the methods used to accomplish our solution. In this section we expose some aspects of the implementation in more detail such as the reasons behind the use of the new mapping, the implementation of the operations, insights on the new ones, and the branching heuristics used.
Let \( C_1 = (V_1, D_1, K_1) \) and \( C_2 = (V_2, D_2, K_2) \) be two CSPs corresponding to FMs \( M_1 \) and \( M_2 \) respectively. Assuming that every variable in \( V_1 \) corresponds to a variable in \( V_2 \) and vice versa.

1: Post \( C_1 \land \neg C_2 \)
2: if a solution is found then
3: \( \text{return} \ C_1 \) and \( C_2 \) are not equivalent
4: else
5: Post \( C_2 \land \neg C_1 \)
6: if a solution is found then
7: \( \text{return} \ C_1 \) and \( C_2 \) are not equivalent
8: \( \text{return} \ C_1 \) and \( C_2 \) are equivalent

\textbf{Algorithm 5:} Improvement of Algorithm 4

<table>
<thead>
<tr>
<th>Naïve Approach</th>
<th>Non-Naïve Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm 2</td>
<td>Algorithm 3</td>
</tr>
<tr>
<td>Test 1</td>
<td>7,752.90</td>
</tr>
<tr>
<td>Test 2</td>
<td>7,375.50</td>
</tr>
<tr>
<td>Test 3</td>
<td>—</td>
</tr>
<tr>
<td>Test 4</td>
<td>—</td>
</tr>
</tbody>
</table>

Average case
Worst case

Tests could not be run for the reasons discussed in Section 8.2.

\textbf{Table 4:} Runtimes (in seconds) of the different tests of the equivalence operation, using different implementations.

\subsection*{6.2 The New Mapping}

In Section 4.2 we described the basics of the mapping proposed in [1], and the modification we made to work with in this project. In this section we explain the idea behind this new mapping and the results we obtained when using it.

\subsubsection*{6.2.1 Motivation}

The mapping rules of [1] require all implicit attribute \textit{selected} to be boolean decision variables meaning that their domain is \{false, true\} (or \{0,1\}),
while in our mapping we define the attribute selected of the optional features as boolean decision variable whereas the attribute selected of the mandatory features are defined as decision variables with domain \{true\} (or \{1\}).

This eliminates the necessity of posting the constraints of all of the mandatory relations between mandatory features and the optional relations in which the parent feature is a mandatory one. Moreover, the constraints of type X requires Y and X excludes Y where X is a mandatory feature, can be replaced for Y = true and Y = false respectively.

On the other hand, constraints such as “total power consumption of the hardware parts cannot exceed the capacity of the power supply” required the CP model to have sets of constraints of the form:

\[
\begin{align*}
\text{CPU1.c} & \in \{0, 62, 76, 99\}, \\
\text{CPU1} & \Rightarrow \text{CPU1.c} \in \{62, 76, 99\}, \\
\#\backslash \text{CPU1} & \Rightarrow \text{CPU1.c} \neq 0
\end{align*}
\]

where CPU1 represents the implicit attribute selected of the feature CPU1, and CPU1.c its cost. The same had to be done for every feature whose cost, memory consumption, or power consumption had to be modeled. The new mapping diminishes the effort required to model these situations for the mandatory features, reducing the constraints shown above to a single one:

\[
\text{CPU1.c} \in \{62, 76, 99\},
\]

6.2.2 Outcome

The idea behind the new mapping appeared very promising for the models we worked with because we were able to eliminate a lot of constraints. We ran the operations filter and, number of products on the computer, and the package CP models with 3 different branching heuristics to compare their performances. The runtimes displayed in Table 5 show a significant improvement on the operations tested on the computer CP model, with a gain of up to 14%. However the highest gain on the package model was of 5%. More on this is discussed in Section 7.2.

Nevertheless, the reduction on the number of constraints increases the developer productivity by making the coding of the CP model easier (we will revisit this matter in Section 8.3), and the program more readable at the same time.
<table>
<thead>
<tr>
<th>Heuristic</th>
<th>Filter</th>
<th>N. of Prod.</th>
<th>Filter</th>
<th>N. of Prod.</th>
</tr>
</thead>
<tbody>
<tr>
<td>leftmost-step</td>
<td>7.38</td>
<td>6.38</td>
<td>11.56</td>
<td>10.04</td>
</tr>
<tr>
<td>ffc-step</td>
<td>11.58</td>
<td>10.84</td>
<td>9.56</td>
<td>8.52</td>
</tr>
<tr>
<td>leftmost-enum</td>
<td>7.30</td>
<td>6.38</td>
<td>11.70</td>
<td>10.06</td>
</tr>
</tbody>
</table>

| Mapping proposed in [1] | New mapping |

Table 5: Runtimes for comparing the mapping proposed in [1] versus the new mapping

6.3 The Analysis Operations

6.3.1 Overview

The analyses over FMs are performed by different operations. The purpose of each operation is to answer a question regarding the FM providing useful information to its stakeholders. In this study we have tested a total of seventeen operations, listed in Section 3.2, on the cellphone, computer, and package FMs. Twelve (1–12) were taken from the implementation of the computer FM provided by the authors of [2] in which it was presented, and later modified and reused to decrease their runtimes, adapt them to the cellphone and package FM, and make them scalable. To the best of our knowledge we were the first to introduce the next four (13–16), and to present algorithms and experimental results for the equivalence operation (17), even though some studies claim to support it [3].

6.3.2 The New Operations

Thanks to the CP approach and tools we used, coding the newly introduced operations on an FM, namely: core attributes, variant attributes, dead attributes and attributes description was very straightforward. In general this was a matter of the following three simple steps:

1. Posting the constraints of the corresponding CP model.

2. Branching on the variables of the CP model representing the attributes of the FM.

3. Labeling them the as core, variant, or dead depending on their domains.
The equivalence operation on the other hand required more work, considering that a negation of the CP model had to be coded into the solver, and that it was the heaviest one with respect to the runtime. Regardless, the operation was implemented (as explained in Section 5.6) and tested on the models as well.

The runtimes of (all) the operations performed on both the computer and the package FM are given in Section 7. We refrain from showing runtimes for the phone FM due to the fact that they are all too small (∼0s).

6.4 Best Branching Heuristic

The behaviour of the branching heuristics is analysis and instance dependent, hence the only way to find the best one for a specific analysis and instance is with experimental trials. Tables 5–8 show how the combinations leftmost-step, and leftmost-ffc work better for the filter and number of products operations on the computer CP model. However for the operations on the package model, leftmost-enum beats all of the other heuristics.

It is important to notice that even though branching heuristics can influence the performance of the solver in general, the improvements made in the analyses of our CP models were not big enough to make it worth to experiment with different orderings to find the best one. After all, running an operation with two different heuristics to compare them takes longer than running it with the default.

We believe that the reason why different heuristics have such small impact on analyses on FMs, is that the decision variables of the corresponding CP models are either boolean decision variables, or integer decision variables with small domains. This also makes it harder conjecture about what would be the best ordering. However, based on the reason stated above if a tool for automation like the one proposed in Section 8.3 were to be developed, we would suggest a variable and value ordering resembling the behaviour of SICStus Prolog’s ffc-enum.
### Table 6: Runtimes (seconds) for filter operation on the computer CP model

<table>
<thead>
<tr>
<th>Method</th>
<th>leftmost</th>
<th>min</th>
<th>max</th>
<th>ff</th>
<th>ffc</th>
</tr>
</thead>
<tbody>
<tr>
<td>step</td>
<td>6.32</td>
<td>8.56</td>
<td>6.58</td>
<td>10.48</td>
<td>10.86</td>
</tr>
<tr>
<td>enum</td>
<td>6.36</td>
<td>8.26</td>
<td>6.68</td>
<td>10.08</td>
<td>10.16</td>
</tr>
<tr>
<td>bisect</td>
<td>6.38</td>
<td>8.42</td>
<td>6.64</td>
<td>10.54</td>
<td>10.80</td>
</tr>
</tbody>
</table>

*default

### Table 7: Runtimes (seconds) for number of products operation on the computer CP model

<table>
<thead>
<tr>
<th>Method</th>
<th>leftmost</th>
<th>min</th>
<th>max</th>
<th>ff</th>
<th>ffc</th>
</tr>
</thead>
<tbody>
<tr>
<td>step</td>
<td>9.54</td>
<td>9.80</td>
<td>9.78</td>
<td>9.74</td>
<td>8.26</td>
</tr>
<tr>
<td>enum</td>
<td>9.84</td>
<td>10.28</td>
<td>10.24</td>
<td>10.06</td>
<td>8.50</td>
</tr>
<tr>
<td>bisect</td>
<td>9.86</td>
<td>10.20</td>
<td>10.14</td>
<td>10.08</td>
<td>8.42</td>
</tr>
</tbody>
</table>

### Table 8: Runtimes (seconds) for filter operation on the package CP model

<table>
<thead>
<tr>
<th>Method</th>
<th>leftmost</th>
<th>min</th>
<th>max</th>
<th>ff</th>
<th>ffc</th>
</tr>
</thead>
<tbody>
<tr>
<td>step</td>
<td>514.14</td>
<td>577.16</td>
<td>569.04</td>
<td>540.78</td>
<td>571.460</td>
</tr>
<tr>
<td>enum</td>
<td>473.18</td>
<td>574.52</td>
<td>577.02</td>
<td>527.14</td>
<td>563.760</td>
</tr>
<tr>
<td>bisect</td>
<td>520.34</td>
<td>579.76</td>
<td>577.18</td>
<td>541.00</td>
<td>565.100</td>
</tr>
</tbody>
</table>

### Table 9: Runtimes (seconds) for number of products operation on the package CP model

<table>
<thead>
<tr>
<th>Method</th>
<th>leftmost</th>
<th>min</th>
<th>max</th>
<th>ff</th>
<th>ffc</th>
</tr>
</thead>
<tbody>
<tr>
<td>step</td>
<td>788.10</td>
<td>867.76</td>
<td>869.22</td>
<td>834.38</td>
<td>863.94</td>
</tr>
<tr>
<td>enum</td>
<td>745.46</td>
<td>872.30</td>
<td>874.68</td>
<td>817.52</td>
<td>869.96</td>
</tr>
<tr>
<td>bisect</td>
<td>795.70</td>
<td>871.84</td>
<td>879.10</td>
<td>844.26</td>
<td>872.66</td>
</tr>
</tbody>
</table>

#### 7 Critical Analysis of the Solution

#### 7.1 Highlights

In this section we intend to present our opinion regarding how well the problem was solved.
Using CP enabled us to translate FMs into CSPs and thereafter into a series of SICStus Prolog programs in a very simple manner. Such programs are easy to follow even if one is not familiar with the solver or the Prolog programming language because the syntax is very intuitive. Moreover, the results of not using ad-hoc algorithms or data structures for the FMs but instead using a tool designed to model a great variety of real-life problems, are CP models that are very flexible towards modification which is a key aspect of the SPL methods.

On the downside, there is no tool that we know of that would automatically convert FMs into SICStus Prolog, which makes the task of coding them a bit tedious, especially when working with large FMs. The implementation of such tool is suggested as future work in Section 8.3.

7.2 Benefits of the New Mapping

The new mapping proposed in Section 5.2 was used in the experimental trials of the analysis operations. It showed significant improvements on the computer CP model but the impact on the package model wasn’t as promising.

We believe that there are two reasons why the performance did not improve as much on the larger model. First, all of the constraints that were eliminated were of the form (or similar to): Computer $\leftrightarrow$ Hardware, $(CPU2 \land CPU2.speed = 1600) \Rightarrow (CPU2.c = 62 \land CPU2.p = 18)$, or $MB \Rightarrow MB.p \in \{53\}$, thus their propagation would take almost no time; and second, the propagators would have run at most twice before they are subsumed.

Nonetheless, developer productivity would definitely improve along with code readability if an FM is coded into a solver using this mapping given that we eliminated 269 constraints from the original 573 (47%) of the computer model, and 292 out of 608 (48%) on the package model.

7.3 The Operations

The CP approach made it possible for this work to be one of the studies that have support for a considerably high amount of analysis operations. We were able to test 17 analysis operation on 3 different CP models, four of which were introduced in this work. We also presented several approaches to implement the equivalence operation along with the runtimes of the experiments tried
on the computer and package CP models. Furthermore our experimental results prove that these operations scale well for large FMs.

7.4 Performance of the Operations

The operations we have presented in this work were tested on the cellphone, the computer, and the package FMs, shown in Figures 1, 2, and 3 respectively. The runtimes of the last two are displayed in Table 10. The runtimes of the operations on the phone FM are too small (∼0s) which is why we abstain from showing them.

We ran the test on a Debian GNU/Linux 5.0 machine with the following specifications: Processor Intel® Core™2 Duo CPU P8400 @ 2.26GHz, Memory size 4GB, L2 cache 3 MB.

8 Critical Analysis of the Work

Constraint programming has been successfully applied to real-life problems in a wide range of areas such as air traffic scheduling, optimization problems, molecular biology, electrical engineering, among others [13]. One of our main motivations for taking on this project is that the problem we are dealing with is also real, and with CP we would have the opportunity to use the theory of a rising technology as we study the behaviour of the solution in practice.

8.1 Contributions

In our opinion the most important contributions of this work are the introduction of four analysis operations, and the proposal for the equivalence operation along with the experimental results shown, not only for these five but for all seventeen operations on which we ran tests.

Other contributions of this work include increasing development productivity especially when coding the CSP equivalent of an FM into a CP solver, pointing out the advantages and drawbacks of the CP approach, exposing difficulties one might encounter and offering ways to overcome them, placing our solution in contrast with others, illustrating characteristics of the SICStus Prolog solver, and showing how it all works in practice; all to achieve the ultimate goal, exhibiting an elegant solution for automated reasoning on FMs.
<table>
<thead>
<tr>
<th>Analysis Operations</th>
<th>Computer CP Model</th>
<th>Package CP Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Void feature model</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Valid product</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Valid partial configuration</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>All products</td>
<td>6.22</td>
<td>821.68</td>
</tr>
<tr>
<td>Number of products</td>
<td>8.26</td>
<td>745.46</td>
</tr>
<tr>
<td>Commonality</td>
<td>8.76</td>
<td>745.28</td>
</tr>
<tr>
<td>Filter</td>
<td>6.32</td>
<td>473.18</td>
</tr>
<tr>
<td>Core features</td>
<td>0.04</td>
<td>0.26</td>
</tr>
<tr>
<td>Variant features</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>Dead features</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>False optional features</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>Optimization (maximize cost)</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>Optimization (minimize cost)</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>Core attributes</td>
<td>0.24</td>
<td>0.24</td>
</tr>
<tr>
<td>Variant attributes</td>
<td>0.24</td>
<td>0.24</td>
</tr>
<tr>
<td>Dead attributes</td>
<td>0.24</td>
<td>0.24</td>
</tr>
<tr>
<td>Attributes description</td>
<td>0.26</td>
<td>0.26</td>
</tr>
<tr>
<td>Equivalence (equiv. models)</td>
<td>1 0.02</td>
<td>2 82.80</td>
</tr>
<tr>
<td>Equivalence (not equiv. models)</td>
<td>3 0.02</td>
<td>4 0.02</td>
</tr>
</tbody>
</table>

1 Algorithm 5, Test 1.  
2 Algorithm 5, Test 2.  
3 Algorithm 5, Test 3.  
4 Algorithm 5, Test 4.

**Table 10:** Operations’ runtimes (seconds) in computer and package CP models
8.2 Difficulties

During the elaboration of this project we came across some challenging tasks that we had to overcome in order to accomplish our goal.

From the research point of view, the first and probably the most important one is the lack of FMs that are available in the bibliography and on the web. It is very difficult to find real-life FMs that are large and complex enough that would make it interesting to run analyses on them. In addition to this, only a handful of papers present empirical results when using constraint programming to reason on FMs.

From the developer perspective, the built-in predicate when trying to find all of the solutions of a goal in the Prolog programming language is `findall/3`. This predicate stores all the solutions in the list that it receives as its third parameter. As a result when we implemented the all products operation for the first time and ran it in the package FM, which has a total of 86,765,568 of products, we rapidly ran out of memory. The solution to this particular problem was to output a solution as soon as it was found and then continue computing the rest.

Memory consumption was also an issue when we tried to derive full and partial configurations on the FM for the same reason. To resolve this, we used Prolog’s failure driven loops [18] combined with a counter that stores information relevant to the operation in which it is being used.

Another difficulty arose during the testing of the naïve algorithms for the equivalence operation on the larger models, because we had to compare two (non-ordered) large lists of solutions for equality. Checking for each solution of one list if it was present in the other, led us to the same memory consumption issue explained above. Another thing worth mentioning is that the SICStus Prolog global constraints are not reifiable, and implementing custom propagators for them is beyond the scope of this thesis, thus we had no choice but to decompose them into binary constraints when building the logical negation of the CP models, in order to implement the non-naïve approach of the equivalence operation.

8.3 Future Work

We believe that by pursuing this goal we have also opened doors for future work on this matter. Such work could be based for instance on direct improvements of the operations shown in this one, the introduction of new
operations relevant to SPLs that can be solved using CP, or a comparison of
the performance of the operations on different CP solvers.

Another suggestion for future work would be developing a tool that pro-
vides a new level of abstraction between FMs and CP solvers. This is because
FMs are often very large so it becomes laborious to code their corresponding
CP models into the solvers. For instance the computer FM became a 400
lines program.

The tool could provide automated translation from a specification or
markup language such as Z or XML, however the FM would have to be
coded in such language which would require roughly the same effort as man-
ually coding the corresponding CP model of an FM. Thus, it would be ideal
if such tool could be equipped with a graphical interface that supports de-
signing of FMs, and translating to a CP solver, therefore eliminating the
need of manually translating the FM into a CP model, and at the same time
detaching the analysis from a specific solver.

9 Comparison With Related Work

During our research on the reasoning methods proposed for FMs we came
across several works that engaged to address this matter, some of them with
a more similar approach to ours than others. In this section we intend to
contrast this project with previous studies done in this area. We focus on
the works [1, 2, 3, 4, 12, 16], leaving aside those in which a non-CP approach
(such as using BDD or SAT solver) was used.

9.1 First Proposals and Mapping

The first proposals to use CP to reason on FM were made in [12, 4, 3] along
with a method for translating FMs into a generic CSP. In [1] a new mapping
is introduced which may include complex feature-feature, feature-attribute,
and attribute-attribute cross-tree relationships. A minor modification of the
latter was used in this project.

9.2 Operations, Sample Feature Models, and CP Solvers

The definitions for the operations can be found in [2, 3, 4]. The authors of [2]
implemented the first twelve analysis operations listed in Section 3.2 on the
computer FM using the mapping they proposed in [1]. Their experimental results showed improvements gained by the use of global constraints on the computer CP model.

In [4, 12, 14] the authors gave performance results for the void, number of products, all products, optimization, commonality, filter, and variability factor operations. In [17] only void and number of products were executed. void was also a part of [16] where the authors also gave corrective explanations for FMs. Last, multi-step configuration analysis was implemented in [15].

Some studies used real-life FMs and for some, randomly generated FMs were the core of the analyses. We modified and reused all 12 operations described in [2] and made them scalable and faster. In addition, we introduced four and implemented one more to make a total of 17. All of them were tested on the phone, computer, and package CP models to study the performance on models of difference sizes and to evaluate the scalability of the operations.

In this thesis and [2] the CP Solver used was SICStus Prolog as opposed to some other tools that were used in previous studies which are shown in Table 2.

9.3 A Final Word Regarding Improvements

As we mentioned in previous sections, the authors of [1, 2] provided the FMs they work with in both papers for us to study, and try to extend and improve (in terms of the operations). After extensive experiments we were able to lower the runtimes for most of the heavy operations using the new mapping, and branching heuristics different from the default, even more than they already had with the use of global constraints. Furthermore, we reduced the runtime of their heaviest operation (i.e. commonality) eliminating close to half its computation. In addition to that we modified the operations so they would perform well in larger CP models making them scalable, with the use of a failure driven loop and a counter or simply outputting the solutions, instead of storing them in a list in memory.

What’s more, we were able to successfully implement the four operations introduced in this work and the equivalence operation for the computer and package CP models.

We extend our gratitude to them once again for giving us the means and the opportunity to extend some of their work.
References


