

Using Microsoft Solver Foundation to Analyse Feature Models and Configurations

Juan Carlos Navarro
Universidad Piloto de Colombia
juan-navarro@unipiloto.edu.co

Jaime Chavarriaga
Universidad de los Andes
Vrije Universiteit Brussel
jchavarr@vub.ac.be

ABSTRACT

Feature Models are widely used in Software Product Lines to represent commonalities and variabilities in a family of products and to support the interactive configuration of these products. They comprise features and options that can be included in a product and constraints about which combinations of features are allowed. Libraries and frameworks such as SPLOT and FaMa help engineers to determine if a feature model is valid, detect errors in the models and validate if a configuration (i.e., a set of features selected by an user) does not contradict the constraints in the model. Regrettably, these libraries are based on Java and cannot be used in .Net platforms such as the recent Windows Phone systems. For that platforms, there is a Microsoft Solver Foundation (MSF) library that provides a set of solvers such as the used to analyse feature models. This paper explains (1) how to translate feature models and configurations into Constraint Satisfaction Problems in MSF, and (2) how to use that library to determine if a model is valid, enumerate all the valid configurations, and detect core and dead features that may exist in the model. In addition, we present a performance evaluation of the approach.

Categories and Subject Descriptors

D.2.13 [Reusable Software]: Domain engineering

General Terms

Design

Keywords

Feature Model, Constraint Programming, Microsoft Solver Foundation

1. INTRODUCTION

Feature Models (FMs) are the de-facto standard for documenting, checking and reasoning about the configurations of a software system [5]. They are used in Software Product Lines (SPLs) to represent the common and optional (variable) features that may comprise a product. Domain Engineers use these models to understand the set of products and design the shared assets (e.g., components) that will be

used to create all of them. In addition, engineers use these models to create product configurators that validate if the features selected by an user can be used to find or to build a valid product.

Multiple approaches have been defined to analyse feature models [2] and determine if the models are correct and do not include any error. For instance, Benavides et al. [4][3] presented an approach that convert feature models into a Constraint Satisfaction Problem (CSP) and use a Constraint Programming Solver (CP) to detect errors and perform some analyses. Karatas et al. [6][7] extended that approach to support extended feature models and to introduce a larger set of operations. Slightly modified versions were later presented by Alvarez [1], and Mazo et al. [10]. However, existing literature describes implementations of these operators in systems such as SICStus¹ and SWI prolog² and frameworks such as SPLOT³ and FaMa⁴ that works on Java, but not on frameworks based on .Net platforms.

This paper presents our implementation of operations to analyse feature models and configurations using Microsoft Solver Foundation (MSF)⁵, a .Net library aimed to specify, simulate and solve different types of mathematical and constraint models. Our implementation uses existing theories about Feature Models and Constraint Programming to translate the models into MSF-based models and use the mentioned library to solve the model and implement analysis operations.

In this paper, we describe (1) how to translate a feature model into a MSF-based CSP model, (2) how to use the MSF API to implement analysis operations, and (3) how our approach performs when processing models with different sizes.

The remainder of this paper is structured as follows. Section 2 introduces feature models, the operations defined to analyse these models, existing approaches to implement these operations and the MSF. Section 3 details how to translate a FM into a MSF-based CSP and how to implement the mentioned operations. Finally, Section 4 discusses a performance evaluation of this approach, and Section 5 concludes the paper.

¹<http://sicstus.sics.se/>

²<http://www.swi-prolog.org/>

³<http://www.splot-research.org/>

⁴<http://www.isa.us.es/fama/>

⁵<https://msdn.microsoft.com/en-us/devlabs/hh145003>

2. BACKGROUND

This section gives an overview of the feature models used in Software Product Lines, the operations that have been defined to analyse these models, the use of CSP to implement these operations and the Microsoft Solver Foundation.

2.1 Feature Models

A *Feature Model* specifies the similarities and differences among the members of a family of products as well as the options that can be used to configure each product [5]. In Software Product Lines, these models have been used to determine which elements or components can be included in a product.

Feature Models.

A **Feature Model** depicts the features of a product and their dependencies [13]. Typically, it is represented by a hierarchical structure: a tree with a root feature known as the *concept* (i.e., the product to configure in the tree), a set of *child features* that conforms the branches and leaves, and a set of *feature groups* and *feature relationships* representing configuration constraints.

Table 1 shows the elements that may comprise a feature model. Basically, in a feature model:

- the *Root* feature represents the *concept* depicted in the model,
- *Mandatory features* represent commonalities, features that must be selected when the parent feature is already selected,
- *Optional features* capturing variations, features that may be selected or not when the parent is selected,
- *Or (inclusive-or) groups*, where one or more features can be selected, and
- *Alternative (exclusive-or) groups*, where just one of the set can be selected.

Feature relationships are:

- *requires* relationships that indicates that one feature must be selected when other is selected, and
- *excludes* representing that two features cannot be selected at the same time.

Figure 1 shows an example feature model for Cellular Phones. The feature model has the root feature of *Cellular Phone* which has three mandatory subfeatures: *LCD*, *Input Device* and *Battery* and one optional subfeature of *External Memory*. In turn, an *LCD* can be *Normal* or *Touch Screen* but not both. The *Input Device* can be a *Keypad*, an *Stylus*, or both. There is a *excludes* relationships indicating that when a cell phone includes an *Stylus*, cannot include a *Normal LCD* (it must include a *Touch Screen*). Finally, the *Battery* can be of *Small Size* or *Large Size*. There is also a *requires* relationship denoting that the use of a *Touch Screen* requires the inclusion of a *Large Size Battery*.

Feature Configurations.

A **Feature Configuration (or Configuration)** is a set of features of a FM. A configuration is *valid* if the set of features satisfies the constraints defined in the model. For instance, considering the above feature model, a valid configuration for a *Cellular Phone* is $C = \{ CellularPhone, LCD, Normal, InputDevice, Keypad, Battery, SmallSize \}$

Note that not all the combinations of features are valid configurations. For instance, if a configuration includes *LCD*






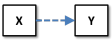
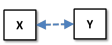
Element	Graphic	Description
root		Concept or Product to configure. It must be selected.
mandatory		if X is selected, Y must be selected
optional		if X is selected, Y can be selected (or not)
or group		if X is selected, one or more of Y_1, \dots, Y_n must be selected
alternative		if X is selected, only one of Y_1, \dots, Y_n must be selected
requires		if X is selected, Y must be selected
excludes		if X is selected, Y must be not selected

Table 1: Feature Model elements

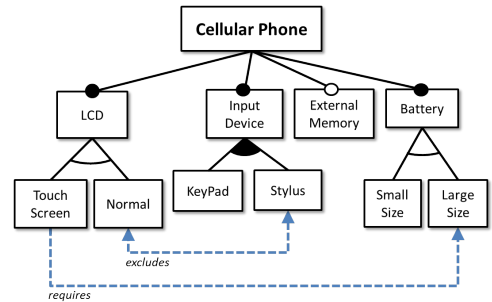


Figure 1: Example Feature Model, adapted from [8]

Normal and *LCD Touch Screen*, it is invalid because the model states that the user must select one of them but not both. In addition, if a configuration does not include neither *Keypad* or *Stylus*, the configuration is invalid because the user must select an *Input Device* with at least one of *Keypad* or *Stylus*.

2.2 Automatic Analysis of Feature Models

In real SPLs, feature models may grow to hundreds or thousands of features [11][9]. In such models, detecting problems and conflicts becomes a hard task for humans.

Analysis Operations.

Some researchers have focused on defining operations to analyse feature models and answer questions that may be of use to the stakeholders of a product [2]. For instance, an analysis operation is aimed to determine if a model is

valid (i.e., if it is not void). Considering that feature models represent the features for a product, a feature model is valid if it represents at least one product. Sometimes the model has errors or conflicts that prevent the definition of any set of features that satisfies the constraints of the model. For instance, if the model states that some feature is mandatory but also defines that the same feature cannot be selected, the model is not valid because you cannot define a configuration (a set of features) that includes that features and excludes that feature at the same time.

Other analysis operations aim to *detect errors* in the feature models. For instance, there is an operation to detect *dead features*, i.e., features that cannot be included in any product, and *core features* (a.k.a., *full mandatory* features), i.e., the set of features included in all the products.

A Catalog of Analysis Operations.

Benavides et al. [2] defined a set of operations to analyse feature models that includes:

- **Validating a Feature Model** (or detecting FM satisfiability), i.e., detecting if at least one product is represented by the FM.
- **Validating a Product**, i.e., detecting if a set of features is valid regarding the features and constraints defined in the FM.
- **Validating a Partial Configuration**, i.e., detecting if a set of features does not include any contradiction regarding the FM.
- **Obtaining all products**, i.e., retrieving all possible products of a FM.
- **Calculating the number of products**
- **Obtaining core features** (or full-mandatory features), i.e., finding the features that appear in all the valid configurations.
- **Obtaining dead features**, i.e., finding the features that never appear in any valid configuration.
- **Obtaining variant features**, i.e., finding the features that may appear or not in a valid configurations.

2.3 Constraint Programming

In order to implement these analysis operations, multiple approaches have been defined. Some of these approaches translate each FM into a CSP and rely on automatic solvers to find the answers.

Constraint Satisfaction Problems.

A **Constraint Satisfaction Problem (CSP)** is a mathematical problem defined in terms of variables, possible values for that variables (i.e., domains) and constraints. A solution for the problem comprises a set with a value for each variable that satisfy the constraints.

For instance, if we define two variables a and b values in the domains $D_a = D_b = \{x \in \mathbb{Z} | x = 0 \vee x = 1\}$ (i.e., that only may contain 0 or 1), and with a constraint $a > b$, a solution (the only one) is $\{a = 1, b = 0\}$. Sometimes the constraints in the problem cannot be satisfied by any combination of values, in such cases we say that the problem is *unfeasible*.

A **Constraint Programming (CP) solver** is a software able to take an specification of a CSP and, if the problem is feasible, to determine a solution. Usually, these solvers can determine the total number of solutions and the values for each solution. In addition, some solvers offer options to

define decisions and goals and determine the best solution based on a given criterion.

2.4 Microsoft Solver Foundation

Microsoft Solver Foundation (MSF) is a .Net library that can be used to solve CSPs and optimization models. Developers use MSF to create models by specifying variables, constraints, goals and data, and execute automatic solvers to find the corresponding solutions. MSF includes some built-in solvers for linear, nonlinear and constraint programming. In addition, it can be used with other third-party solvers.

In MSF, models can be specified using scripts in the Optimization Modeling Language (OML), spreadsheets in Excel, functional programs in F# or imperative programs in any other .Net language. For instance, using C#, a CSP model can be created by creating objects that represent the variables and constraints. Additional objects representing the Solver engine can be used to solve the model and obtain the results.

MSF Services.

MSF Services is an API that allow developers create models independently of the type of solver to use. Using this API, the program must define the variables as *Decisions* and invoke a *solve* method specifying which type of solver use. Listing 1 shows an excerpt of a C# program that define and solve a CSP model using MSF Services. Note that the program invokes the *solve* using a *ConstraintProgrammingDirective* object.

```
// using Microsoft.SolverFoundation.Services; 1
2
SolverContext context = 3
    SolverContext.GetContext();
Model model = context.CreateModel(); 4
5
Domain values = Domain.IntegerRange(0,1); 6
7
Decision a = new Decision(values, "a"); 8
Decision b = new Decision(values, "b"); 9
10
model.AddDecisions( a, b ); 11
model.AddConstraints("constraints", a > b ); 12
13
Solution solution = context.Solve(new 14
    ConstraintProgrammingDirective());
while (solution.Quality != 15
    SolverQuality.Infeasible)
{ 16
    Report report = solution.GetReport(); 17
    Console.WriteLine("{0}", report); 18
    solution.GetNext(); 19
} 20
```

Listing 1: Sample code using MSF Services

MSF CSP solver.

Instead of creating a generic model that can be used with any solver, it is possible to create a model that will work exclusively with a CSP solver. Listing 2 shows an excerpt of a C# program that define a model to be solved exclusively with a CSP solver.

```

1 // using Microsoft.SolverFoundation.Solvers;
2
3 ConstraintSystem model =
4     ConstraintSystem.CreateSolver();
5
6 CspDomain values =
7     model.CreateIntegerInterval(0, 1);
8 CspTerm a = model.CreateVariable(values, "a");
9 CspTerm b = model.CreateVariable(values, "b");
10 model.AddConstraints(model.Greater(a, b));
11
12 ConstraintSolverSolution solution =
13     model.Solve();
14 while (solution.HasFoundSolution)
15 {
16     Console.WriteLine("a : {0}",
17         solution.GetIntegerValue(a));
18     Console.WriteLine("b : {0}",
19         solution.GetIntegerValue(b));
20     solution.GetNext();
21 }

```

Listing 2: Sample code using the MSF CSP Solver

MSF OML.

In addition, MSF supports the *Optimization Modeling Language (OML)*, a domain specific language to specify models. Using OML, the developer can define the model using a more concise syntax. Listing 3 shows the same example in OML.

```

1 Decisions[Integers[0, 1],
2     a,b
3 ],
4 Constraints [
5     a>b
6 ]

```

Listing 3: Sample model defined in OML

A program can read an OML model instead of creating objects representing the variables and constraints. The process to solve the model is similar to the used to process a model using the MSF Services API.

2.5 CSP to analyse Feature Models

Many authors have proposed implementations based on CSPs for the analysis operations presented before: Benavides et al. [4][3] were the first transforming a feature model into a CSP and using a CP solver to validate and perform analyses. Karatas et al. [6][7] extended that approach to support extended feature models and to introduce a larger set of operations. Modified versions were later presented by Alvarez [1] and Mazo et al. [10] to support more types of features and constraints.

Basically, these approaches transform the elements and relationships in the feature model into a CSP. They usually take features and relationships and transform them into variables and constraints. For instance, considering the above example FM for a Cellular Phone, the options for the LCD includes a Normal LCD and a Touch Screen LCD. Each of these features may be translated into the variables *Normal* and *TouchScreen* that will take a value of 0 when the feature is not-selected and 1 when the user select the corresponding option, i.e., $D_{Normal} = D_{TouchScreen} = \{x \in Z | x = 0 \vee x = 1\}$. In addition, because these features belong to an *xor-group*, the user can select only one of these features, i.e., they can be translated to $ifLCD = 1 \implies Normal + TouchScreen = 1$

3. ANALYSING FEATURE MODELS WITH MICROSOFT SOLVER FOUNDATION

We have implemented analysis operations for feature models using MSF. This section presents (1) how we translate feature models into a CSP, and (2) how we implement the analysis operations.

3.1 Translating FMs into CSPs

Feature Models represents features that may be selected by an user to specify a product and constraints among which features can be selected at the same time. These options and constraints can be translated to an constraint programming. Table 2 summarizes the mappings we use (based on the proposed by Benavides et al. [4][3] and van der Broek [14]).








Element	Graphic	CSP
root		$r = 1$
mandatory		$x = y$
optional		$x \geq y$
or group		$x \rightarrow \sum y_i \geq 1$
alternative		$x = \sum y_i$ (i.e., $x \rightarrow \sum y_i = 1$)
requires		$x \leq y$
excludes		$x + y \leq 1$

Table 2: Feature Model mappings to CSP

The mapping between a FM and a CSP has the following general form:

- each feature is mapped to a variable in the CSP;
- the domain for each variable is the same: $\{0, 1\}$, where 0 is used to represent non-selected features and 1 for a feature selected by the user; and
- types of features, groups and relationships are represented as constraints.

The elements of the model are transformed to constraints as following:

- **Mandatory features:** let X be the parent and Y the child in a mandatory relationship, then the equivalent constraint is $Y = X$
- **Optional feature:** let X be the parent and Y the child in an optional relationship, then the equivalent constraint is $X \geq Y$
- **Or group:** let X be the parent of an alternative group and Y_1, \dots, Y_i the set of children, then the equivalent constraint is $if(X = 1, \sum Y_i \geq 1)$

- **Alternative (xor) group:** let X be the parent of an or-group and Y_1, \dots, Y_i the set of children, then the equivalent constraint is $X = \sum Y_i$
- **Requires:** Consider the relationship $X \xrightarrow{\text{requires}} Y$, the equivalent constraint is $X \leq Y$
- **Excludes:** Consider the relationship $X \xrightarrow{\text{excludes}} Y$, the equivalent constraint is $X + Y \leq 1$

For instance, consider the FM presented in Figure 1. It can be modeled in MSF using the CSP-specific API, such as the depicted in Listing 6. First, it is necessary to create the solver and define the domain for the variables. Instead of creating a new domain, we use the *DefaultBoolean* domain that is equivalent to *CreateIntegerInterval(0, 1)*. Then, the program must define the variables representing each feature. And Finally, it must include into the model the constraints representing the feature groups and relationships.

3.2 Implementing Analysis Operations

Once Feature Models are translated to a CSP, Analysis Operations can be implemented by using a CP solver.

Validation.

The validation of a feature model is straightforward. A *valid* feature model requires that a concrete configuration can be derived from the specification. Considering that we transform the FM into a CSP, the FM is valid if a CP solver can find a solution for the CSP. The FM is *valid* if the corresponding CSP is *feasible*. In contrast, the FM is *void* if the CSP is *unfeasible*.

```

1 public boolean IsValid(ConstraintSystem model)
2 {
3     ConstraintSolverSolution solution =
4         model.Solve();
5     return solution.HasFoundSolution;
}

```

Listing 4: Sample code to validate a model

All products.

The set of *all products* of a feature model is the set of valid configurations regarding that model. In MSF, this operation is straightforward. The CP solver included in MSF can return all the solutions of a CSP.

```

1 public List<Configuration>
2     GetAllProducts(ConstraintSystem model)
3 {
4     List<Configuration> list = new
5         List<Configuration>();
6     ConstraintSolverSolution solution =
7         model.Solve();
8     while (solution.HasFoundSolution)
9     {
10        list.Add(new Configuration(solution));
11    }
12    return list;
13 }

```

Listing 5: Sample code to determine all the products

Number of products.

The *number of products* of a feature model is the total number of valid configurations regarding that model. This

```

// using Microsoft.SolverFoundation.Solvers;
ConstraintSystem model =
    ConstraintSystem.CreateSolver();

// define the domain
CspDomain values = model.DefaultBoolean;
// define the variables
CspTerm CellPhone =
    model.CreateVariable(values, "CellPhone");
CspTerm LCD = model.CreateVariable(values,
    "LCD");
CspTerm TouchScreen =
    model.CreateVariable(values,
    "TouchScreen");
CspTerm Normal = model.CreateVariable(values,
    "Normal");
CspTerm InputDevice =
    model.CreateVariable(values,
    "InputDevice");
CspTerm KeyPad = model.CreateVariable(values,
    "KeyPad");
CspTerm Stylus = model.CreateVariable(values,
    "Stylus");
CspTerm ExtMemory =
    model.CreateVariable(values, "ExtMemory");
CspTerm Battery = model.CreateVariable(values,
    "Battery");
CspTerm SmallSize =
    model.CreateVariable(values, "SmallSize");
CspTerm LargeSize =
    model.CreateVariable(values, "LargeSize");

// add the constraints
model.AddConstraints(
    // Root
    model.Equal(1, CellPhone),

    // LCD is mandatory
    model.Equal(CellPhone, LCD),
    // LCD or-group
    model.Equal(LCD,
        model.Sum(TouchScreen, Normal)),

    // Input Device is mandatory
    model.Equal(CellPhone, InputDevice),

    // Input Device alternative group
    model.Implies(
        model.Equal(1, LCD),
        model.GreaterThan(
            model.Sum(TouchScreen, Normal),
            1)
    ),

    // ExtMemory is optional
    model.GreaterThanOrEqual(CellPhone,
        ExtMemory);

    // Battery is mandatory
    model.Equal(CellPhone, Battery),
    // Battery or-group
    model.Equal(LCD,
        model.Sum(SmallSize, LargeSize)
    );

    // Normal LCD excludes Stylus
    model.LessThanOrEqual(Normal, Stylus);

    // Touch Screen requires
    // a Large Size Battery
    model.LessThanOrEqual(
        model.Sum(TouchScreen, LargeSize),
        1);

```

Listing 6: Sample code to create the CSP for the sample FM

can be easily obtained in MSF by requesting that value to the solver.

Core Features.

A *core feature* is a feature which is part of all the products described by the feature model. Core Features cannot be disabled because of relations and/or constraints in the model. This means that, if we add to the CSP an assumption (a constraint) disabling a core feature, i.e., setting the value of 0, the modified model will result *invalid/infeasible*.

```

1 public List<Feature>
  GetCoreFeatures(List<Feature> features,
  ConstraintSystem model)
2 {
3   List<Feature> list = new List<Feature>();
4   foreach (Feature f in features)
5   {
6     CspTerm constraint = model.Equal(0, f)
7     model.AddConstraints(constraint);
8     ConstraintSolverSolution solution =
        model.Solve();
9     if (!solution.HasFoundSolution)
10    {
11      list.Add(f);
12    }
13    model.RemoveConstraints(constraint);
14  }
15  return list;
16 }

```

Listing 7: Sample code to determine all the core-features

Dead Features.

A *dead feature* is a feature which cannot be part in any product although it is part of the feature model. Dead features occur because of constraints in the model which force them to be disabled. In consequence, if we add to the CSP an assumption selecting a dead feature, i.e., setting the value of 1, the modified model will result *invalid/infeasible*.

```

1 public List<Feature>
  GetDeadFeatures(List<Feature> features,
  ConstraintSystem model)
2 {
3   List<Feature> list = new List<Feature>();
4   foreach (Feature f in features)
5   {
6     CspTerm constraint = model.Equal(1, f)
7     model.AddConstraints(constraint);
8     ConstraintSolverSolution solution =
        model.Solve();
9     if (!solution.HasFoundSolution)
10    {
11      list.Add(f);
12    }
13    model.RemoveConstraints(constraint);
14  }
15  return list;
16 }

```

Listing 8: Sample code to determine all the dead-features

Variant Features.

A *variant feature* is a feature that can be enabled or disabled in one product of the represented by the feature model. Basically, a feature can either be *core*, *dead* or *variant*. The

set of variant features can be determined after computing the set of *core* and *dead features*.

```

public List<Feature>
  GetVariantFeatures(List<Feature> features,
  ConstraintSystem model)
1
2 {
3   List<Feature> cores =
  GetCoreFeatures(features, model);
4
5   List<Feature> deads =
  GetDeadFeatures(features, model);
6
7   return features.Except(cores)
  .Except(deads).ToList();
8
9 }

```

Listing 9: Sample code to determine all the variant-features

Valid Product (or Valid Configuration).

A *Valid Product* is a configuration (i.e., a set of features) that defines completely a product and satisfies all the constraints defined in a feature models. Basically, this *complete configuration* includes all the features for a product. Those features not included must be disabled. To check if a product is valid, we add to the CSP assumptions enabling each selected feature, i.e., setting them the value of 1, and disabling each non-selected feature, i.e., setting the value of 0. If the resulting CSP is feasible, the product is valid. If not, the product is not valid.

Valid Partial Product (or Valid Partial Configuration).

A *Valid Partial Product* is a configuration that is not complete but does not contradict any constraint defined in the feature model. Basically, is a configuration where not-included features are not disabled. The partial product is valid if the resulting CSP is feasible,

```

public Boolean
  IsValidConfiguration(Configuration config,
  List<Feature> features, ConstraintSystem
  model, bool partial = false)
1
2 {
3   foreach( Feature f in features)
4   {
5     if (config.Contains(f))
6     model.AddConstraints(model.Equal(1,f));
7     else
8     if (!partial)
9     model.AddConstraints(model.Equal(0,f));
10  }
11  ConstraintSolverSolution solution =
  model.Solve();
12  return solution.HasFoundSolution;
13 }

```

Listing 10: Sample code to validate a configuration

4. EVALUATION

We have implemented a .Net library to analyse feature models⁶, using MSF and supporting the diverse mappings and operations presented above. In our tests, this library was able to process the feature models correctly with a performance similar to other existing libraries. This section discusses the tests and performance evaluation we made.

⁶<http://github.com/FaMoSA/fma.net>

Model	# feat	CTRC	Valid (ms)	# prods	All Prods (ms)	1st 50k (ms)	# dead	Dead (ms)
PUJ WebStore	17	0	20	816	32	-	0	81,2
Dell Computers	47	76	21,4	2319	83	-	0	134,8
FrasCaTi 1.4	63	57	21,2	+1 billion	-	1331,2	0	159,4
J2EE Arch	77	0	21	+1 billion	-	1544,0	0	158,8
XText	137	1	14,8	+1 billion	-	2144,4	0	320,0
Printers	172	0	21,2	+1 billion	-	3148,0	0	371,4

Table 3: Data obtained processing some selected SPLOT feature models

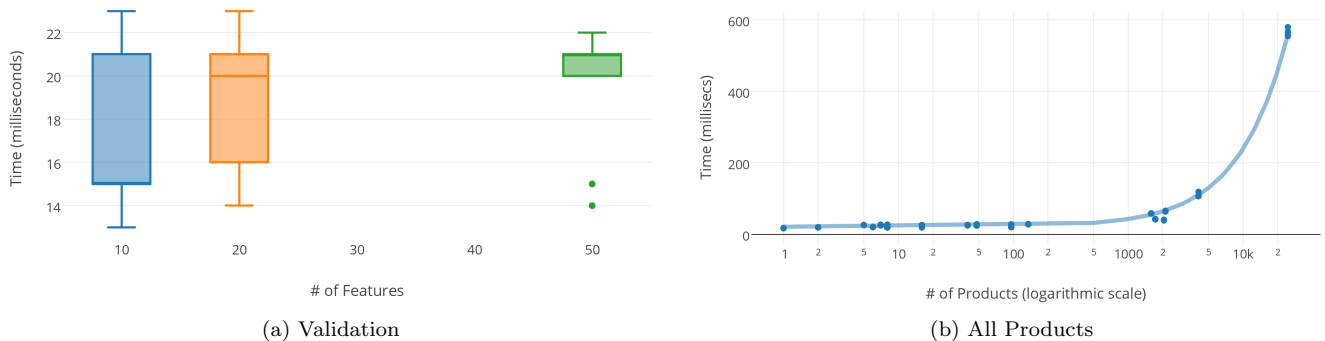


Figure 2: Performance of processing some FeatureIDE feature models

4.1 Tests

In order to test our implementation, we choose publicly available feature models. We use models included in the FaMa Test Suite⁷ to check if the library is able to detect valid and invalid feature models. In addition, we took some models with different sizes from the FeatureIDE test suite⁸ and the SPLOT repository⁹. These models were also used to evaluate the library’s performance.

4.2 Performance Evaluation

Here we present two set of models used in our evaluation. On one hand, we used a set of feature models from the SPLOT repository: PUJ WebStore, Dell Computers, FrascaTi, J2EE Arch, Printers, and Xtext. On the other hand, we took features models from the FeatureIDE test suite. This test suite has thousands of models with different sizes. We used thirty models: three sets of 10 models with 10, 20 and 50 features.

We executed a set of tests on a Intel Core i7-2600 4.4GHz computer with 16GB RAM running Windows 7 64 bits. We measured the time using the standard *System.Diagnostics* classes existing in .Net. Here we present data of executing the operations five times for each model.

Validation of a Feature Model.

We test the operation that validates a feature model using both sets of models. Table 3 shows the time spent to validate the models from SPLOT. The table shows the number of features of the model (in the “# feat” column) and the time of processing in milliseconds. Figure 2a shows the per-

⁷http://www.isa.us.es/fama/?FaMa_Test_Suite

⁸http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/

⁹<http://www.splot-research.org/>

formance executing the operation with models of different sizes. The X axis shows the number of features and the Y axis the time of processing in milliseconds. Note that the times are quite similar among them. According to an study of Pohl et al. [12], the performance of the MSF solver is similar to the observed with other solvers in C and Java.

All Products.

To validate a model, our library must invoke the solver one time. To get all the products, it is necessary to invoke the solver as many times as the number of products. Table 3 shows the number of products of each SPLOT model and the time spent. Note that the first two models represents 816 and 2319 products while the others represent more than 1 billion each one (according to SPLOT). For the last four models, there is the time spent to determine the first 50000 products.

Figure 2b shows the performance of the “all products” operation according to the number of products. The X axis shows the number of products using a logarithmic scale and the Y axis the time of processing in milliseconds. Note that the time of processing increases as the number of products increase.

Dead Features.

In contrast to the “all products” operation, the operation to detect all the “dead features” invokes the solver one time for each feature. Other operations, such as “core features” and “variant features” exhibit a similar behaviour. Table 3 shows the time spent to detect the dead features in the selected SPLOT models. Note that the time increase as the number of features increase. Figure 3 shows the performance of the “dead features” operation with models of different sizes. Note that the processing time increases as the number of features increase.

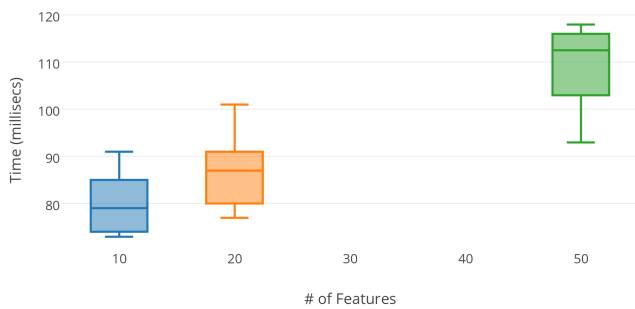


Figure 3: Performance of obtaining Dead Features in some FeatureIDE models

5. CONCLUSIONS

We have presented our implementation of operations to analyse feature models and configurations using Microsoft Solver Foundation (MSF). It supports the complete set of analysis operations described by Benavides et al. [2] and can be used in .Net applications instead of libraries and frameworks such as SPLOT or FaMa. This is very useful in platforms that may run .Net applications but not Java libraries.

We have performed a complete set of tests to ensure that operations works correctly. In addition, we have performed a performance evaluation included in this paper.

Our implementation does not include some optimizations proposed by other authors. For instance, there are approaches that combine solvers [11], eliminates non-relevant features [15], and use heuristics to simplify the model [9][12]. Implementing these optimizations remains in the future work.

There are other transformation operations on features models (e.g., merge and slice) not mentioned in this paper. In addition, there other operations to support interactive configuration (e.g., to autocomplete or recommend options to users). Currently, we are working on implementing these operations using Microsoft Solver Foundation and other .Net-based solvers.

6. ACKNOWLEDGMENTS

Jaime Chavarriaga is a recipient of a COLCIENCIAS fellowship. Part of this work has been supported by a cooperative project between Universidad de los Andes and Siemens Colombia.

7. REFERENCES

- [1] C. E. Alvarez. Automated Reasoning on Feature Models via Constraint Programming. Master's thesis, Uppsala Universiteit, Sweden, 2011.
- [2] 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.
- [3] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. Using Java CSP Solvers in the Automated Analyses of Feature Models. In *Generative and Transformational Techniques in Software Engineering (GTTSE 2005)*, pages 399–408. Springer Berlin Heidelberg, 2006.
- [4] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated reasoning on feature models. In *Advanced Information Systems Engineering (CAISE 2005)*, pages 491–503. Springer Berlin Heidelberg, 2005.
- [5] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) feasibility study. (CMU/SEI-90-TR-021). Technical report, Software Engineering Institute, Carnegie Mellon University, 1990.
- [6] A. S. Karataş, H. Oğuztüzün, and A. Dođru. Mapping extended feature models to constraint logic programming over finite domains. In *14th International Conference on Software Product Lines (SPLC'10)*. Springer-Verlag, 2010.
- [7] A. S. Karataş, H. Oğuztüzün, and A. Dođru. From extended feature models to constraint logic programming. *Science of Computer Programming*, 2012.
- [8] S. Kim, D.-K. Kim, L. Lu, and S. Park. Quality-driven architecture development using architectural tactics. *Journal of Systems and Software*, 82(8):1211 – 1231, 2009.
- [9] J. H. J. Liang, V. Ganesh, K. Czarnecki, and V. Raman. Sat-based analysis of large real-world feature models is easy. In *19th International Conference on Software Product Line, (SPLC 2015)*, pages 91–100, 2015.
- [10] 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'11)*. Springer-Verlag, 2011.
- [11] M. Mendonca, A. Wařowski, and K. Czarnecki. Sat-based analysis of feature models is easy. In *13th International Software Product Line Conference (SPLC '09)*, pages 231–240, 2009.
- [12] R. Pohl. Improving the performance of the automated analysis of feature models by applying graph width measures. Master's thesis, University of Duisburg-Essen, Essen, Germany, 2012.
- [13] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature diagrams: A survey and a formal semantics. In *14th IEEE International Conference Requirements Engineering (RE 2006)*, pages 139–148, 2006.
- [14] P. van den Broek. Optimization of product instantiation using integer programming. In *14th International Software Product Line Conference (SPLC 2011)*, pages 107–111, 2011.
- [15] H. Yan, W. Zhang, H. Zhao, and H. Mei. An optimization strategy to feature models' verification by eliminating verification-irrelevant features and constraints. In *11th International Conference on Software Reuse, (ICSR 2009)*, pages 65–75, 2009.