Framework for automated analyses of feature models

Content

What's FaMa? 4
FaMa flavors 4
Installation 4
FaMa Shell 5
OSGi 5
Web Service (WSDL) 5
Standalone 5
Quick start 6
FaMa Shell 6
OSGi 6
Web Service (WSDL) 7
Standalone 8
Deeping FaMa 8
Background 8
Models 10
Feature models 10
Extended feature models 11
Input file formats 11
XML format 11
Plain text format 12
Operations 14
Validation 14
Products 14
Number of products 15
Commonality 15
Variability 15
Valid product 16
Valid configuration 16
Error detection 17
Error explanations 17
Invalid product explanation 18
Core features 18
Variant features 19

Reasoners 19
Choco reasoner 19
Sat4j reasoner 20
JaCoP reasoner 20
JavaBDD reasoner 21
Selection criterions 21
Transformations 21
FaMa Shell 21
FaMa standalone 23
Basics 23
QuestionTrader 23
Working with models 24
Config file 25
FaMa OSGi 25
FaMa web service 26
Extending FaMa 26
Troubleshooting 26
Contact us 27
Further work 28
Acknowledgements 28
Glossary 28
References 29

What's FaMa?

FaMa (FeAture Model Analyser) is a tool to analyse feature models. And what's a feature model? Think you are going to buy a new car. When you are going to buy a new car, you can configure a lot of features of it: colour, lights, power… A feature model is a way to represent all these features (or others) in a single model.

And what can FaMa do for me? Think again in the previous example. Now you are the vendor, and you need to know how many different car configurations are. Or maybe you are the buyer, and you want to know what configuration is the cheapest. Doing these tasks manually is a tedious and error-prone task. However, FaMa is able to do these tasks and others automatically. You have to provide only a feature model, and select an operation for it.

More formally, FaMa is a tool to analyse Software Product Lines, represented as feature models. Software product line is a recent software development paradigm, based on the principle of "customization in mass". But how we have presented, FaMa has uses further than analysing software product lines strictly. FaMa can analyse any system that can be expressed as a set of features on a hierarchy, with or without attributes.

FaMa flavors

You can use FaMa by four ways. We provide a shell front-end to final users, and three ways to integrate your own application with FaMa: a java standalone version, a SOAP/WSDL web service, and a set of OSGi bundles.

- **FaMa shell**: is the current front end for final users. A command-line interface where you can load models, and invoke analysis operations.
- **FaMa Web Service**: we provide a WSDL with most of FaMa operations. You can consume it for your application.
- **FaMa OSGi**: FaMa is also available as a set of OSGi bundles. OSGi (www.osgi.org) is a java specification for services integration.
- **FaMa standalone**: moreover, FaMa is available as an extensible library.

Installation

Now, we show you how to install FaMa in its four forms.

First steps you have to do (except on FaMa WS) are to go to www.isa.us.es/fama/, fill the request form and download it. You have to make sure too you have installed last version of Java Runtime Environment (version 6). You can check it entering "java –version" command on a command line/shell. If not, go to www.java.com and install it.

If you have problems when you have finished these lines, go to "Troubleshooting" section. If problems persist, send us an email ("Contact" section).

FaMa Shell

Steps:

1. Using a shell/console, enter on the FaMa folder (previously created from the zip file), and type "java –jar lib/FaMaSDK-1.1.1.jar" to execute main jar of FaMa.
2. If everything is all right, you should see FaMa prompt ($).

OSGi

Steps:

1. Get an OSGi distribution. There are several OSGi distributions. We recommend you Equinox (integrated with eclipse) or Felix. Both are the most stable and mature implementations of OSGi standard. If you want do OSGi subsection of QuickStart, you should use Equinox.

1.1. If you want to use FaMa through Equinox on Eclipse:

   a. Get an eclipse distribution at www.eclipse.org if you don't have it. If you don't know what version to download, we recommend you Eclipse classic. You have enough information about installing and using eclipse on this site.
   b. When installed, copy all FaMa jars (on root directory and lib folder) to "dropins" eclipse folder.
   c. Launch eclipse, and open Plug-in registry view (Window/ Show view/ Other/ Plug-in development/ Plug-in registry). Make sure jars you have copied on previously step are installed and started. If they aren't started, right click on it, "Show advanced operations". Then, right click again, and "Start".

1.2. If you want to use through Felix:

   a. Go to http://felix.apache.org and download last Felix version (Felix Framework Distribution). If you already have a Felix distribution, make sure you delete "felix-cache" folder to avoid problems.
   b. Decompress it to a folder.
   c. Enter on Felix folder, and copy FaMa jars to "bundle" folder.
   d. Start Felix from command line: enter on Felix folder, and type "java –jar bin/felix.jar". You should see a new prompt.
   e. Type "lb" to check FaMa bundles are installed and started.

Web Service (WSDL)

Steps:

1. WSDL address is http://150.214.188.147:8081/ADAService?wsdl. Take a quick glance, and make sure the site is online.
2. Use a web service tool, as Apache Axis for java, or an IDE that provides WSDL tools, as Eclipse for java or Visual Studio for .NET. These tools can create stub classes to use a web service. If you are using Eclipse:
   a. Install (if don't) Axis tools for eclipse. Open eclipse/ *Help/ Install new software*. On "Working with", select the name of your eclipse distribution, and when screen loads all plugins, go to *Web, XML and Java EE Development/Axis2* Tools and select it.

b. Create an empty project
c. File/ New/ Other/ Web Services/ Web Service Client. Paste WSDL location on the upper box, and select assemble (second level) on the left. Next and finish. Stub classes and imported libraries are now on the project.

Standalone

Steps:

1. Create an empty project, and copy to it FaMaSDK jar (adding to project libraries), lib folder (don't add to project libraries), and FaMaConfig.xml to project's root directory.
2. Add the library to the project. If you use Eclipse:
   a. Add libraries to the project through buildpath options. Right click on jar file/ Build Path/ Add to Build Path.

Quick start

We are going to explain a quickstart example for the four forms to use FaMa. This example consists of load a feature model from a file (HIS.fm), and doing two analysis operations (validation and number of products).

FaMa Shell

Steps:

1. Enter on the FaMa Shell with this command (you should be at FaMa dir): `java -jar lib/FaMaSDK-1.1.1.jar`. Now you are on an interactive shell session. If you type `help` you can see commands to use.
2. Use "load" command to load a feature model from a file. Inside folder *fm-samples* you have many model examples. Load *HIS.fm* file with `load fm-samples/HIS.fm`. An OK message should appear. Once model a model is loaded, you can invoke analysis operations over it.
3. First, we are going to invoke a validation operation. Type `valid` command. Result should be "Model is valid".
4. Now that we know our model is valid, we want to know how many products this model has. Type `#products` and see the result. It should be *32*.
5. You may invoke more operations if you want. Type help to see them. Type `help nameOfTheCommand` to see a detailed description of the command. When you finish, you can leave the shell typing `exit`.

OSGi

To use FaMa through an OSGi implementation, you have to implement an OSGi bundle that consumes FaMa services offered. We are going to teach you how to do this using *Eclipse* IDE, and its OSGi implementation, *Equinox.*

Steps:

1. Start eclipse, and create a new Plug-in project: File/ New/ Other/ Plug-in Development/ Plug-in Project. On "This plug-in is targeted to run with", select Equinox, and click next. On Execution environment, click next, and finish.
2. When project has been created, go to the MANIFEST file, and on "Dependencies", add a required plug-in: es.us.isa.FaMaSDK. Save it.
3. Now, go to the Activator class. On start method, you should get services you want (QuestionTrader on our case), and on stop method, you should release these services (you should save service references you get). Your start method may look as follows.

```
public void start(BundleContext context) throws Exception {
     String className = QuestionTrader.class.getCanonicalName();
     //attribute sr (ServiceReference) defined on the Activator
     sr = context.getServiceReference(className);
     QuestionTrader qt = (QuestionTrader) context.getService(sr);
     if (qt != null){
          System.out.println("FaMa load successful");
          //here you use FaMa as on Standalone version
          consumeFaMa(qt);
     }
}
```

Method `consumeFama` has to contain similar code to code presented on *Standalone* subsection.

And your stop method:

```
public void stop(BundleContext context) throws Exception {
     if (sr != null){
         context.ungetService(sr);
     }
}
```

4. Finally, run the project as an OSGi application. Select "Run configurations". For workspace bundles, select only this project, and for Target platform, unselect all (click on Target platform box), and then click on "Add requires bundles". Then select every fama bundle (all es.us.isa packages), apply and run. Output should be

```
FaMa load successful
The number of products is: 4.0
```

Web Service (WSDL)

To consume FaMa web service, make sure your internet connection is OK, and FaMa WSDL is available on http://150.214.188.147:8082/FaMaWS?wsdl. If you have created stub classes with Eclipse and Axis tools, use this code to implement the example of this section.

```
FaMaWSPortType fama = new FaMaWSPortTypeProxy("http://localhost:8082/FaMaWS");
byte[] model = model2bytes(new File("fm-samples/HIS.fm"));
boolean b = fama.isValid(model);
System.out.println("Is the model valid?"+b);
```

We provide you code for method model2bytes too:

```
private byte[] model2bytes(File f) {
	int size = (int) f.length();
	byte[] res = new byte[size];
	InputStream in;
	try {
		in = new FileInputStream(f);
		in.read(res);
	} catch (FileNotFoundException e) {
		e.printStackTrace();
	} catch (IOException e) {
		e.printStackTrace();
	}
	return res;
}
```

Standalone

Once you have followed installation steps for standalone version of FaMa, you can execute the example presented in this section. Copy this source code to a main method, and run it:

```
//The main class is instantiated
QuestionTrader qt = new QuestionTrader();

//A feature model is loaded
VariabilityModel fm = qt.openFile("fm-samples/HIS.fm");
qt.setVariabilityModel(fm);

//////// VALID QUESTION + NUMBER PRODUCTS QUESTION /////////
ValidQuestion vq = (ValidQuestion) qt.createQuestion("Valid");
qt.ask(vq);
if (vq.isValid()) {
   NumberOfProductsQuestion npq = (NumberOfProductsQuestion)
      qt.createQuestion("#Products");
   qt.ask(npq);
   System.out.println("The number of products is: "+
      npq.getNumberOfProducts());
} else {
   System.out.println("Your feature model is not valid");
}
```

You can find this code on folder *fama-samples* too.

Console should show this output text:

```
The number of products is: 4.0
```

Deeping FaMa

Background

Software Product Lines (SPL) arises as an approach for software mass customisation. SPLs pursue to meet as many individual customer's needs as possible with the fewest possible effort. SPLs mainly focus on analysing the commonalities and variabilities among the software products the potential customers could demand. Variability models (VMs) are the artefacts used to represent the common and variable parts in a SPL.

FAMA is a tool to analyse variability models. There exist several kinds of variability models, being Feature Models (FMs) the most used. A FM represents all the products that can be built in a SPL in terms of features. A feature is an observable characteristic of a product. Features are hierarchically structured. The root feature in the top of the structure represents the overall functionality of a product. It is refined in child features which represent more specific functionalities of the product. Features are refined in child features step by step forming a tree-like structure. Since a FM remark the variant features, parent and child features are connected by means of relationships of different kind. The most common relationships are mandatory, optional and cardinality-sets, although some others are proposed in the bibliography. The tree-like structure can be broken using cross-tree constraints to remark exclusions and dependences among features that cannot be set in the same branch of the tree.

Sometimes features are not enough to represent the possible combinations of features, needing to represent more complex information. Extended FMs add attributes to features to represent information such as versioning, development time and cost, memory consumption, etc.

FAMA is able to analyse VMs to extract relevant information to support decision making during SPL development. Examples of analysis operations are which are the products that can fit into a customer need; which is the cheapest product and its cost or which are the errors within a FM.

Why is FAMA built? The rhythm at which we produced research results was faster that our ability to produce software proof-of-concepts. Since our development budget was very reduced, researchers had to assume development tasks so we had to reduce the prototyping time at a maximum so research activities were not compromised. We needed for a flexible tool that offered a set of services that permit a researcher to incorporates his/her results in a minimal time. Existing tools had no public license and their extensibility was very limited.

We designed a reference architecture to integrate the results at date and the future researches we could bear in mind. We invested most of our technical sta
budget to build such infrastructureand in one year we had a first version. Since then, 6 prototypes or labs tools have been produced by researchers with few intervention of technical sta. We decided to release the tool in a LGPL v3 license. You can download it at www.isa.us.es/fama.

Properly speaking, FAMA is not a tool but a SPL of VM analysis tools. Each new functionality is considered as a variant feature in the SPL.

FAMA functionalities have been applied in different contexts:

- Variability Modeling Tools: Incorporating analysis capabilities to visual editors and other CASE tools that use VMs somehow. Moskitt Feature Modeler [8], an Eclipse-based visual editor of feature models is an example of it.
- Product Configurators: Providing product configuration capabilities to validate feature selections and to assist end-user by propagating user decision and suggesting

corrections for invalid configurations. These products usually use a fixed feature model. FAMA Debian Package and ISA Packager use FAMA FW for this purpose.

- Dynamic Systems Reconfiguration: Restoring and reconfiguring dynamic systems such as smart homes [9] and TV broadcasting systems [10] whenever errors happen or system preferences change. FAMA Lite for Smart Homes supports the decision of the features to activate or deactivate services whenever a new feature is deployed or an existing one fails.
- Fast Prototyping Framework: Building tools for the development of new variant features for FAMA PL. We have developed FAMA Benchmarking, FAMA Test Suite and FAMA Random Generator to test the functionality and performance of FAMA FW reasoners and FAMA SDK, an environment to develop new variant features.

Models

FaMa works with Feature Models. A feature model is a data structure, similar to a tree, but with more complex relationships. We can use these models to represent product lines in a compact way, like the example presented on Quick Start section. Now, we are going to deep into standard feature models and into extended feature models too.

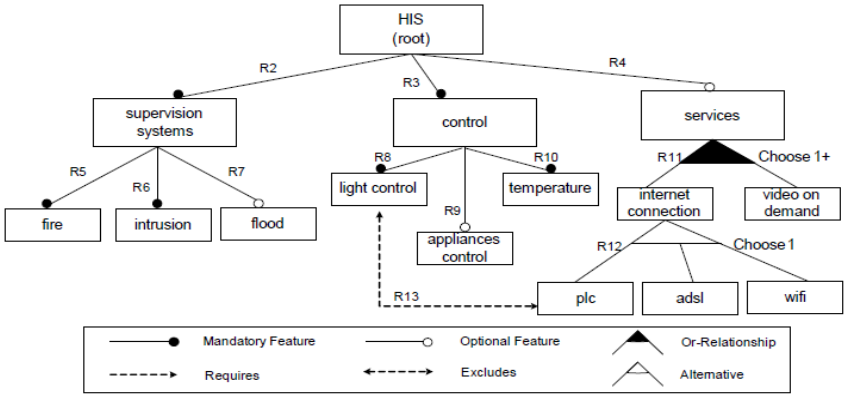
Feature models

As you can see at the image (HIS feature model), a feature model is very similar to a tree. It has a root, and except this node (feature since now), every feature has a parent. Each feature may have three types of relationships with their children:

- Mandatory relationship (1 to 1): if parent is present in a product, child must be too
- Optional relationship (1 to 1): if parent is present in a product, child may be or not.
- Group relationship (1 to n)
  - Set relationship: if parent is present in a product, one (and only one) child must be present.
  - Or relationship: if parent is present in a product, one or more children must be present.
  - Cardinality-based relationship: if a parent is present in a product, user can define how many children must be present.

Moreover, there is a special type of relationship between non parent-child nodes. These relationships, called *Cross Tree Constraints*, are from one feature to another feature (1 to 1). There are two types:

- Excludes: if A excludes B, B and A can't be at the same time in a product.
- Requires: if A requires B, if feature A is present in a product, feature B must be too.

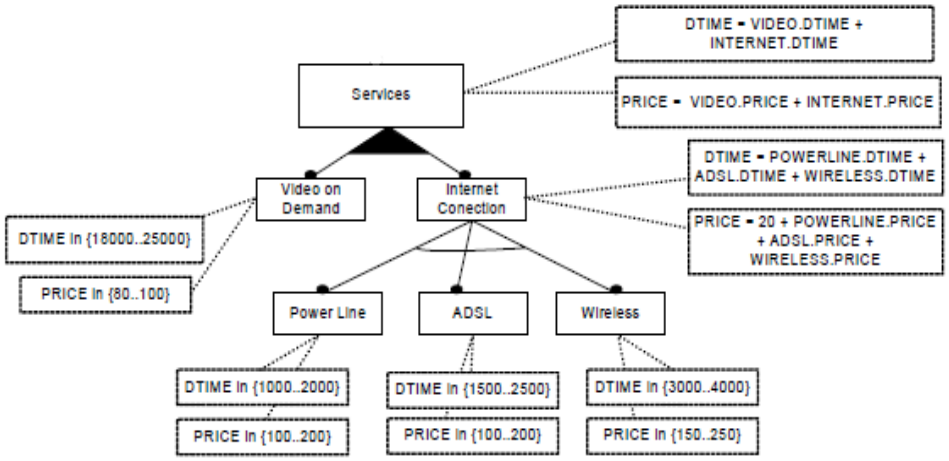


Extended feature models

Extended feature models are standard feature models with attributes. An attributed is related to a feature, and it has a type (integer, real, enumerated…), a domain (set of candidate values), and it may be involved in constraints. These constraints are more complex than Cross Tree Constraints, and they may involve many attributes and features. For instance:

```
Video IMPLIES (Wifi.speed > 54 OR ADSL)
```

You can see below an attributed extension of *Services* branch from HIS feature model. Each feature has two attributes on this sample: dtime and price.



Input file formats

FaMa accepts two input file formats for models: xml and plain text.

XML format

This is the primitive format of FaMa. A XML based format where you can specify features, relationships and their cardinality, and cross-tree-constraints. Several samples with this format and the XML Schema (feature-model-schema.xsd) are attached to FaMa distribution.

```
<?xml version="1.0" encoding="UTF-8"?>
<feature-model xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="http://www.tdg-
seville.info/benavides/featuremodelling/feature-model.xsd">
       <feature name="root">
```

```
            <binaryRelation name="Opt01">
                  <cardinality max="1" min="0" />
                  <solitaryFeature name="child01" />
            </binaryRelation>
            <binaryRelation name="Opt02">
                  <cardinality max="1" min="0" />
                  <solitaryFeature name="child02" />
            </binaryRelation>
      </feature>
</feature-model>
```

Extensions of this format can be *.xml* or *.fama.*

For a deeper description of xml format, you can consult xsd file at http://famats.googlecode.com/files/feature-model-schema.xsd.

Plain text format

Feature Model Format (FMF) is a plain text format where we can define standard or extended feature models.

For basic models, we have two sections.

In `%Relationships`, we define tree structure and relationships types. First defined feature is the root of the feature model. Any feature after colon is a child of the feature before the colon. Name of the feature can contain letters and "_" character. Mandatory relationship is the default one. Optional can be specified with the feature between brackets [], and group relationships with cardinality and a set of features ([n,m]{FeatA FeatB…}).

You can define cross-tree-constraints too in `%Constraints` section (REQUIRES and EXCLUDES among features)

Basic models have *.fm* or *.fmf* file extension. You should use these extensions to avoid errors when using FaMa. You can see here an example of basic feature model in this notation.

```
%Relationships
HIS: SUPERVISION_SYSTEM CONTROL [SERVICES];
SUPERVISION_SYSTEM: FIRE INTRUSION [FLOOD];
CONTROL: LIGHT_CONTROL [APPLIANCES_CONTROL] TEMPERATURE;
SERVICES: [1,2]{VIDEO INTERNET};
INTERNET: [1,1]{POWER_LINE ADSL WIRELESS};

%Constraints
LIGHT_CONTROL EXCLUDES POWER_LINE;
```

For extended models, `%Relationships` section is the same. In `%Attributes`, we define feature's attributes, with their type, domain, default value (default attribute value when feature is selected) and null value (value when feature is not selected). Syntax is as follows:

```
FeatName.AttName: Type[min1 to max1,…, minN to maxN], defaultVal, nullVal;
```

If domain is enumerated, you don't have to put type (as DB.cost in the example).

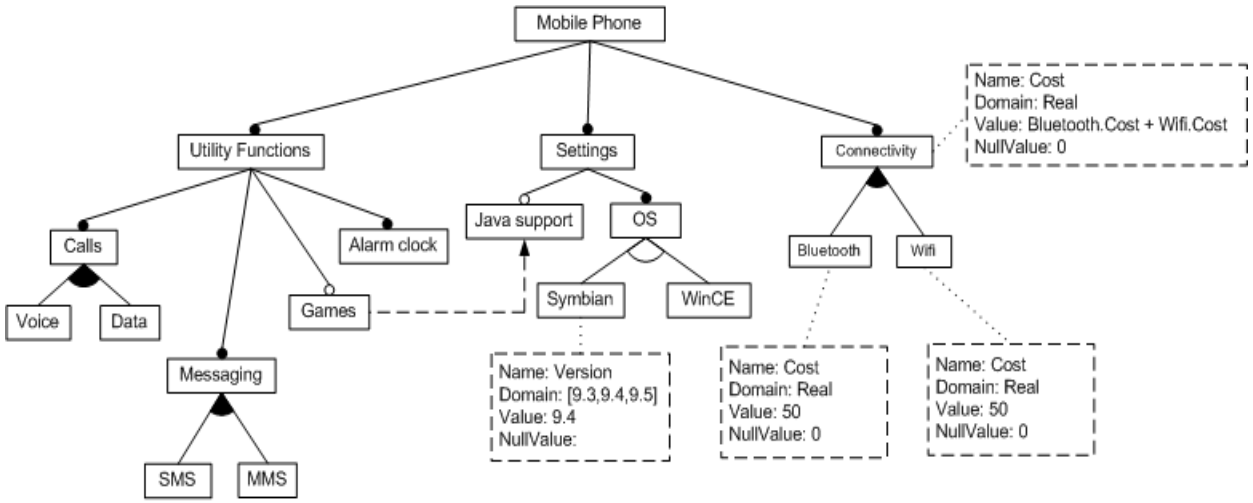
```
FeatName.AttName: [val1, val2,…, valN], defaultVal, nullVal;
```

Finally, on `%Constraints` section, you can merge features and attributes to define complex constraints. You can use arithmetic (+, -, *, /, mod), relational (>, >= , <, <=, ==, !=) and logical (AND, OR, NOT, IMPLIES, IFF (if only if)) operators.

We also provide sugar syntax to define constraints over attributes of a same feature. For instance:

```
FeatureName{
       attName > 0;
}
```

Here we present a full example of an attributed feature model:



- 30% cost reduction in mobile phones including both, bluetooth and wifi connectivity
- Symbian v9.3 is not compatible with wifi connectivity

% Relationships
MobilePhone : UtilityFunc Settings Connectivity;
UtilityFunc : Calls Messaging [Games] Alarm;
Calls : [1,2] {Voice Data};
Messaging : [1,2] {SMS MMS};
Settings : [JavaSupport] OS;
OS : [1,1] {Symbian WinCE};
Connectivity : [1,2] {Bluetooth Wifi};
% Attributes
Symbian.version: [0,3,4,5], 4, 0;
Bluetooth.cost: Integer[0 to 500], 50, 0;
Wifi.cost: Integer[0 to 50], 50, 0;
Connectivity.cost: [0,4],0 , 0;
% Constraints and invariants
Games IMPLIES JavaSupport;
Symbian.version==3 IMPLIES NOT wifi;
Bluetooth and Wifi IMPLIES Connectivity.cost= =(Bluetooth.cost +Wifi .cost);
Connectivity.cost == Bluetooth.cost + Wifi.cost;

Operations

FaMa provides and implements several analysis operations on feature models. Now, we are going to describe them.

Validation

- Description: validates a model. It checks if a model is not empty, or in other words, it has at least one product.
- Input parameters: none
- Output parameters: model validity (Boolean)
- Model support: standard and extended
- Reasoners support: Choco (standard and extended), Sat4j, JavaBDD, JaCoP
- Shell usage: `valid`
- API usage:
  - Id: `valid`
  - Interface: `ValidQuestion`
  - Methods:

Method	Comments
`isValid(): boolean`	Returns if model is valid or not

- Web service usage: `isValid(byte[] model): boolean`

Products

- Description: calculates all valid products of a feature model
- Input parameters: none
- Output parameters: list of valid products
- Model support: standard
- Reasoners support: Choco, Sat4j, JavaBDD, JaCoP
- Shell usage: `products`
- API usage:
  - Id: `products`
  - Interface: `ProductsQuestion`
  - Methods:

Method	Comments
`getAllProducts(): List<? extends GenericProduct>`	Returns the list of valid products
`getNumberOfProducts():long`	Returns number of products

- Web service usage: `getProducts(byte[] model): ProductProxy[]`

Number of products

- Description: calculates the number of products
- Input parameters: none
- Output parameters: number of valid products (long)
- Model support: standard
- Reasoners support: Choco, Sat4j, JavaBDD, JaCoP
- Shell usage: `#products`
- API usage:
  - Id: `#products`
  - Interface: `NumberOfProductsQuestion`
  - Methods:

Method	Comments
`getNumberOfProducts():long`	Returns number of products

- Web service usage: `getNumberOfProducts(byte[] model): long`

Commonality

- Description: calculates appearances of a given feature into the list of products
- Input parameters: a feature of the model
- Output parameters: commonality (long)
- Model support: standard
- Reasoners support: Choco, Sat4j, JavaBDD, JaCoP
- Shell usage: `commonality nameOfAFeature`
- API usage:
  - Id: `Commonality`
  - Interface: `CommonalityQuestion`
  - Methods:

Method	Comments
`setFeature(GenericFeature f):void`	Set the feature to calculate commonality
`getCommonality(): long`	Returns the commonality of the feature

- Web service usage: `getCommonality(byte[] model, String feat): long`

Variability

- Description: calculates variability degree of a feature model

- Input parameters: none
- Output parameters: variability (float)
- Model support: standard
- Reasoners support: Choco, Sat4j, JavaBDD, JaCoP
- Shell usage: `variability`
- API usage:
  - Id: `Variability`
  - Interface: `VariabilityQuestion`
  - Methods:

Method	Comments
`getVariability(): long`	Returns the variability of a feature model

- Web service usage: `getVariability(byte[] model): float`

Valid product

- Description: determines if a product is valid for a given model
- Input parameters: a product
- Output parameters: product validity (boolean)
- Model support: standard and extended
- Reasoners support: Choco (standard and extended), Sat4j, JavaBDD, JaCoP
- Shell usage: `valid-product featureA featureB …`
- API usage:
  - Id: `ValidProduct`
  - Interface: `ValidProductQuestion`
  - Methods:

Method	Comments
`setProduct(Product p):void`	Set the product
`isValid(): boolean`	Returns if the product is valid

- Web service usage: `isValidProduct(byte[] model,ProductProxy p): boolean`

Valid configuration

- Description: analyses if a configuration is valid. A configuration is a non finished product that can need more features to be a valid product.
- Input parameters: a configuration
- Output parameters: configuration validity (Boolean)

- Model support: standard and extended
- Reasoners support: Choco (standard and extended), Sat4j, JavaBDD, JaCoP
- Shell usage: none
- API usage:
  - Id: `ValidConfiguration`
  - Interface: `ValidConfigurationQuestion`
  - Methods:

Method	Comments
`setConfiguration(Configuration p):void`	Set the configuration
`isValid(): boolean`	Returns if the configuration is valid

- Web service usage: none

Error detection

- Description: looks for errors on a feature model
- Input parameters: set of observations (available from the feature model, `getObservations()` method)
- Output parameters: set of errors
- Model support: standard and extended
- Reasoners support: Choco (standard and extended), Sat4j, BDD, JaCoP
- Shell usage: `errors`
- API usage:
  - Id: `DetectErrors`
  - Interface: `DetectErrorsQuestion`
  - Methods:

Method	Comments
`setObservations(Collection<Observation> obs):void`	Set the observations
`getErrors(): Collection<Error>`	Return errors (if the model has them)

- Web service usage (both error explanations): `detectAndExplainErrors(byte[] mode): ErrorProxy[]`

Error explanations

- Description: when a model has errors, this operation look for explanations (relationships) for the errors
- Input parameters: model errors
- Output parameters: model errors (with explanations)
- Model support: standard and extended
- Reasoners support: Choco (standard and extended), JaCoP (standard), Sat4j(standard)
- Shell usage: `errors`

- API usage:
  - Id: `ExplainErrors`
  - Interface: `ExplainErrorsQuestion`
  - Methods:

Method	Comments
`setErrors(Collection<Error> errors):void`	Set the errors
`getErrors(): Collection<Error>`	Return errors (with explanations)

- Web service usage (both error detection): `detectAndExplainErrors(byte[] model): ErrorProxy[]`

Invalid product explanation

- Description: provides options to repair a invalid product for a given model
- Input parameters: a product (invalid)
- Output parameters: a set of selection and deselections (features)
- Model support: standard
- Reasoners support: Choco
- Shell usage: `valid-product`
- API usage:
  - Id: `ExplainProduct`
  - Interface: `ExplainInvalidProductQuestion`
  - Methods:

Method	Comments
`setInvalidProduct(Product p): void`	Set a invalid product
`getSelectedFeatures(): Collection<GenericFeature>`	Return features to be selected
`getDeselectedFeatures(): Collection<GenericFeature>`	Return features to be deselected
`getFixedProduct(): Product`	Return a repair product

- Web service usage: `productRepair(byte[] model, ProductProxy p): ProductProxy`

Core features

- Description: calculates features that are present on every product
- Input parameters: none
- Output parameters: a set of features
- Model support: standard
- Reasoners support: Choco
- Shell usage: `core-features`
- API usage:
  - Id: `CoreFeatures`

- Interface: `CoreFeaturesQuestion`
- Methods:

Method	Comments
`getCoreFeats(): Collection<Product>`	Return core features

- Web service usage: `getCoreFeatures(byte[] model): String[]`

Variant features

- Description: calculates features that are not present on every product
- Input parameters: none
- Output parameters: a set of features
- Model support: standard
- Reasoners support: Choco
- Shell usage: `variant-features`
- API usage:
  - Id: `VariantFeatures`
  - Interface: `VariantFeaturesQuestion`
  - Methods:

Method	Comments
`getVariantFeats(): Collection<Product>`	Return variant features

- Web service usage: `getVariantFeatures(byte[] model): String[]`

Reasoners

A reasoner implements a subset of FaMa operations, using solvers or algorithms. At present, FaMa has four reasoners: choco, sat4j, jacop and javabdd.

Choco reasoner

- Solver: choco
- Paradigm: Constraint Programming
- Link: www.emn.fr/z-info/choco-solver
- Supported models: standard and extended
- Implemented operations:
  - Validation

- Products
- Number of products
- Commonality
- Variability
- Valid product
- Valid configuration
- Detect errors
- Explain errors
- Product explanation
- Core features
- Variant features

Sat4j reasoner

- Solver: sat4j
- Paradigm: SAT (Boolean satisfiability problem)
- Link: www.sat4j.org
- Supported models: standard
- Implemented operations:
  - Validation
  - Products
  - Number of products
  - Commonality
  - Variability
  - Valid product
  - Valid configuration
  - Detect errors
  - Explain errors

JaCoP reasoner

- Solver: JaCoP
- Paradigm: Constraint Programming
- Link: http://jacop.osolpro.com/
- Supported models: standard
- Implemented operations:
  - Validation
  - Products
  - Number of products
  - Commonality

- Variability
- Valid product
- Valid configuration
- Detect errors
- Explain errors

JavaBDD reasoner

- Solver: JavaBDD
- Paradigm: Binary Decision Diagrams
- Link: http://javabdd.sourceforge.net/
- Supported models: standard
- Implemented operations:
  - Validation
  - Products
  - Number of products
  - Commonality
  - Variability
  - Valid product
  - Valid configuration

Selection criterions

When many reasoners implement the same operation, FaMa needs a selection criterion to choose one of them. At this moment, we have available the easiest selection criterion possible: first reasoned we find. We are working on more criterions.

Transformations

A transformation takes as input a model and returns another model equivalent to the first. Transformations can be over the same metamodel, or over different ones.

At now, FaMa provides an atomic set transformation. On an atomic set transformation, all features connected by mandatory relationships are collected into the same feature, improving analysis performance.

FaMa Shell

Commands:

- help

- Usage: `help [nameOfACommand]`
- Description: shows help. If no parameters have been attached, shows a list of available commands. If the parameter is a command, shows the usage of the command.

- load
  - Usage: `load pathToAModel`
  - Description: loads a model from a file. You have to invoke this command before doing any analysis operation.
- exit
  - Usage: `exit`
  - Description: application ends
- commonality
  - Usage: `commonality [feature]`
  - Description: does commonality operation. If no feature is attached, shell will ask for one.
- valid
  - Usage: `valid`
  - Description: does valid operation. If model is not valid, it asks for doing an error detection operation.
- valid-product
  - Usage: `valid-product [featureA feature …]`
  - Description: does valid product operation. If no arguments passed, it asks for them. If product is not valid, it asks for look for a repair for the product.
- errors
  - Usage: `errors`
  - Description: checks for errors. If it finds errors, it asks for look for explanations for them.
- core-features
  - Usage: `core-features`
  - Description: does core features operation.
- variability
  - Usage: `variability`
  - Description: does variability operation.

- #products
  - Usage: `#products`
  - Description: does number of products operation.
- products
  - Usage: `products`
  - Description: does products operation
- variant-features
  - Usage: `variant-features`
  - Description: does variant features operation

FaMa standalone

Basics

FaMa has a facade class, named QuestionTrader. User should interact with FaMa through this class. It provides methods to work with models and operations.

FaMa usage through standalone form has 6 basic steps.

```
//1. Instantiate QuestionTrader
QuestionTrader qt = new QuestionTrader();

//2. Load a model
VariabilityModel fm = qt.openFile("fm-samples/test.fama");

//3. Fix the model
qt.setVariabilityModel(fm);

//4. Create an operation
ValidQuestion vq = (ValidQuestion) qt.createQuestion("Valid");

//5. Execute the operation
qt.ask(vq);

//6. Recover information
boolean b = vq.isValid();
System.out.println(b);
```

QuestionTrader

The next table resumes QuestionTrader methods.

Method syntax	Comments
`addStagedConfiguration(Configuration c): void`	Fixes a configuration
`ask(Question q): PerformanceResult`	Executes an operation
`createQuestion(String s): Question`	Creates a new operation from its id
`createTransform(String s): IVariabilityModelTransform`	Creates a new transformation from its id
`getAvailableReasoners(String questionId): Collection<Reasoner>`	Returns a set of reasoners that implement a chosen question
`getCriteriaSelector(String s): CriteriaSelector`	Returns a CriteriaSelector named as a passed string
`getCriteriaSelectorNames(): Iterator<String>`	Returns an iterator with available criteria selectors
`getHeuristics(String): Map<String,Object>`	Returns a set of available heuristics on FaMa
`getQuestionById(String id): Question`	Returns a question from its id
`getQuestionsId(): Iterator<String>`	Returns all question ids
`getReasonerById(String id): Reasoner`	Returns a reasoned from its id
`getReasonersId(): Iterator<String>`	Returns all reasoned ids
`getSelectedReasoner(): Reasoner`	Returns the fixed reasoned
`getVariabilityModel(): VariabilityModel`	Returns the current variability model
`openFile(String path): VariabilityModel`	Loads a model from a file
`setCriteriaSelector(CriteriaSelector sel): Boolean`	Fixes a CriteriaSelector on FaMa
`setSelectedReasoner(Reasoner r): void`	Fixes a reasoned on FaMa
`setHeuristics(String s): void`	Fixes a heuristic on FaMa

`setVariabilityModel(VariabilityModel v): void`	Fixes a model on FaMa
`writeFile(String path, VariabilityModel v): void`	Writes a model to a file

Working with models

FaMa Works with feature models, and you can do three things with them:

- **Load a model from a file**: loading a model from a file is the most usual way to work with models in FaMa. QuestionTrader's method `openFile(String path)` returns a *VariabilityModel*. *GenericFeatureModel* and *GenericAttributedFeatureModel* are subclasses of this class, and you can cast the model to one of them if you are sure of model's type. Parser used for loading depends on file extension. Xml files uses *.xml* and *.fama*, basic plain text uses *.fm* and *.fmf*, and attributed plain text uses *.afm* and *.efm* extensions.
  Example:

```
VariabilityModel fm = (GenericFeatureModel)qt.openFile("fm-samples/test.fama");
```

- **Save a model to a file:** you may want to save a model into another format, or create from scratch a model a save it into a file. For this purpose QuestionTrader has the method `writeFile(String path, VariabilityModel v)`. File extension determines writer used to save the file.
  Example:

```
qt.writeFile("model.fm", vm);
```

- **Searching and getting features**: for some operations and other tasks, you may need getting one o more features. For this purpose, class *GenericFeatureModel* (extends *VariabilityModel*) has the methods `getFeatures()`: `Collection<GenericFeature>`, that returns all features, and `searchFeatureByName(String name): GenericFeature`, that returns a feature that its name matches the string.
- **Transform a model into another model:** method `createTransform(String s)` returns a IVariabilityModelTransform object. These objects has the method `doTransform(VariabilityModel vm): VariabilityModel` that transforms one model into another model equivalent. By default, FaMa provides one transformation named AtomicSet.
  Example:

```
VariabilityModel vm = qt.openFile("model.fm");
IVariabilityModelTransform vmt = qt.createTransform("AtomicSet");
VariabilityModel tvm = vmt.doTransform(vm);
```

- **Add a configuration**: you can add a configuration over a model, and then execute operations using it. A configuration is a set of selected and deselected features. For instance, for a given a model, a configuration may be *select{A,B} and deselect{D}*, or expressed in other programmatic notation, *{A=1,B=1,D=0}*. *Configuration* class stores this information, and QuestionTrader's method `addConfiguration(Configuration c): void` impose a configuration on

the model. For instance, you can know how many products a model has if featurea A is always selected, and feature B always deselected. For that, you have to create a configuration with these elements, then impose it to the model, and finally create and execute NumberOfProductQuestion:

```
//model previously loaded and imposed
GenericFeature f1 = fm.searchFeatureByName("A");
GenericFeature f2 = fm.searchFeatureByName("B");
Configuration conf = new Configuration();
conf.addElement(f1, 1);
conf.addElement(f2, 0);
qt.addStagedConfiguration(conf);
//now, create and execute an operation, for instance NumberOfProducts
```

Config file

FaMa has a config file named *FaMaConfig.xml*. On this file, we select and configure each FaMa component: reasoners, operations, readers and writers from metamodels, user interfaces…

When you are using FaMa as standalone library, or from the shell, you should make sure that all components are where this file says. Default path is a *lib* folder. This folder is at the same level that FaMaSDK jar.

Configurable elements on FaMaConfig.xml are:

- Reasoner
  - `<reasoner id="reasonerId" file="path/to/jar" class="class.that.implements.Reasoner"/>`
- CriteriaSelector
  - `<criteriaSelector name="name" class="class.that.implements.CriteriaSelector"/>`
- Question
  - `<question id="questionId" interface="interface.that.extends.Question"/>`
- Reader
  - `<reader extensions="fileExt1,…,fileExtN" class="class.that.implements.IReader" file="path/to/jar"/>`
- Writer
  - `<writer extensions="fileExt1,…,fileExtN" class="class.that.implements.IWriter" file="path/to/jar"/>`
- Transform
  - `<transform id="transformationId" interface="class.that.implements.IVariabilityModelTransform" file="path/to/jar"/>`
- UserInterface
  - `<UserInterface file="path/to/jar" mainclass="path.to.main.class"/>`

FaMa OSGi

Once you have created an OSGi bundle to consume QuestionTrader service, there is only a differences between FaMa use through OSGi and the standalone version. FaMaConfig has not

use for FaMa OSGi, since we use OSGi to this task. For any other doubt, you can consult standalone section.

FaMa web service

FaMa WSDL provides many of the operations available on standalone and OSGi versions. Every operation of the WSDL corresponds with one (or more) analysis operation. For each operation, a model should be passed as parameter, since these operations are atomic. Now, we present equivalences between WS operations and standalone operations.

FaMa WS operation	FaMa operation(s)
`isValid`	`ValidQuestion`
`getNumberOfProducts`	`NumberOfProductsQuestion`
`getProducts`	`ProductsQuestion`
`getErrors`	`DetectErrorsQuestion`
`getCoreFeatures`	`CoreFeaturesQuestion`
`getVariantFeatures`	`VariantFeaturesQuestion`
`getVariability`	`VariabilityQuestion`
`getCommonality`	`CommonalityQuestion`
`isValidProduct`	`ValidProductQuestion`
`isValidConfiguration`	`ValidConfigurationQuestion`
`detectAndExplainErrors`	`DetectErrorsQuestion & ExplainErrorsQuestion`
`productRepair`	`ExplainInvalidProductQuestion`

FaMa WS accepts same models and file formats than standalone version. If an operation is available for an attributed model (isValid, for instance), FaMa WS works correctly.

Extending FaMa

You can implement you own reasoner, metamodel or transformation, and integrate it with FaMa (by means of OSGi or FaMaConfig.xml)

Troubleshooting

I can't start OSGi (Equinox/Felix)

Go to website of your OSGi distribution (felix: felix.apache.org, equinox: www.eclipse.org/equinox or knopflerfish: www.knopflerfish.org)

OSGi can't install/start FaMa bundles

Are you sure you have copied all bundles to bundles folder? If you are working on Felix, delete *felix-cache* folder before start again.

Errors appear when starting console

Make sure you are executing FaMaSDK bundle from its folder, and config files (FaMaConfig and commands) are present.

Web service code does not work

It is possible FaMa WS server is off, or you are having problems with your connection. Try later.

Standalone version throws many exceptions when starts, and then crashes (sometimes)

Make sure *FaMaConfig.xml* is present at project root folder and the same for *lib* folder, with metamodels and reasoners inside.

I receive a NullPointerException when I'm using standalone version

NullPointerExceptions are thrown when the most of the time when FaMa isn't able to create an operation. This one can has two causes: name of the operation is misspelled, or you are trying to invoke a non-supported operation for a model. Take a glance into Deeping *FaMa/ Operations* section to check correct spelling and support for operations.

FaMa crashes when loading a model

Maybe the file has not the correct file extension. Make sure that xml files ends with *.xml* or *.fama*, basic plain text files ends with *.fm* or *.fmf*, and attributed plain text files ends with *.afm* or *.efm*.

Other reason may be that basic or extended metamodels are not working properly. If you are using standalone or shell version, revise FaMaConfig.xml file, to assure paths are correct.

If it has a valid extension and FaMaConfig.xml has not errors, maybe file's syntax is failing.

I hate command line front-ends. I want a GUI!!

Don't exasperate. We are working on a integration with MOSKitt, a eclipse based tool to edit graphically feature models.

I've put all standalone libraries on build path, and it crashes again!!

You should NOT put all libraries into build path, put only FaMaSDK library.

The error is not here, and I don't find a solution!!! Help please!!!

Don't worry. Write us an email to fama.support@gmail.com, and we will solve your problem ☺.

Contact us

FaMa website is available on www.isa.us.es/fama

FaMa team is composed by several persons. Now, we are two persons developing the tool: Jose Ángel Galindo and Jesús García (both PhD students). On research level, Fabrizia Roos (PhD student), Pablo Trinidad, David Benavides, Sergio Segura (all PhD) are the "chiefs", under the supreme chief, Antonio Ruiz :-P.

You can contact us at fama.support@gmail.com for more information, bugs, advices, suggestions or what you want.

Further work

Currently, we work on FaMa. We are integrating FaMa with a graphical Feature Model editor (MOSKitt, www.moskitt.org/eng/), and researching and developing new operations, transformations and selector criterions. Visit often www.isa.us.es/fama to stay informed!

Acknowledgements

We want to thank everyone implied in FaMa project, and all users that have given feedback about the tool ☺

Glossary

- Analyser: a FaMa component that uses algorithms or external solvers to implement FaMa operations
- Apache Axis: a java tool to deploy and consume web services.
- Automated analysis: what's FaMa does over feauture models ☺
- Cardinality based relationship: all those relationships that are not mandatory, optional, alternative or "or-relationship". They have a minimum and a maximum number of appearances.
- Criteria selector: a criterion about how to select a reasoned that implements a given operation
- Cross tree constraint: a relationship over two features that aren't parent and child
- Eclipse: an integrated development environment (IDE) for java. www.eclipse.org
- Explanation: set of relationships that cause an error on the model
- Feature: atomic element of a feature model. Represents an asset of the SPL.
- Feature model: a graphical way to represent software product lines, or product lines in general.
- Mandatory: a 1 to 1 relationship. If the parent is present, child has to be present
- Metamodel: a model that defines another model.
- Observation: set of assignments to variables to detect errors on models
- Operation: analysis operation to extract information from a feature model
- Optional: a 1 to 1 relationship which allows the child to be active or not
- Or relationship: a relantioship 1 to n which allows one or more active feature if the parent is present

- OSGi: a framework specification to contain and communicate bundles that offer and consume services.
- Quality attribute: variable attached to a feature, with a name, domain, value and constraints.
- Reasoner: a fama module type. Its task is to implement FaMa analysis operations.
- Set relationship: a relantioship 1 to n which allows only one active feature if the parent is present
- Shell: a text based user interface
- Software Product Line: a set of products that share and reuse assets, process and knowledge.
- Transformation: translation from one model to another equivalent
- Void model: a model with no products

References

- Apache felix: http://felix.apache.org
- Choco solver: www.emn.fr/z-info/choco-solver/
- Equinox: www.eclipse.org/equinox
- FaMa website: www.isa.us.es/fama
- ISA research group: www.isa.us.es
- JaCoP Solver: http://jacop.osolpro.com/
- Java Runtime Environment: http://www.oracle.com/technetwork/java/javase/downloads/index.html
- JavaBDD: http://javabdd.sourceforge.net/
- Sat4j: www.sat4j.org

Software Product Lines: Practices and Patterns. Paul Clements and Linda Northrop